Programmatic Strategy Synthesis: Resolving Nondeterminism in Probabilistic Programs*

KEVIN BATZ, RWTH Aachen University, Germany
TOM JANNIK BISKUP, RWTH Aachen University, Germany
JOOST-PIETER KATOEN, RWTH Aachen University, Germany
TOBIAS WINKLER, RWTH Aachen University, Germany

We consider imperative programs that involve *both* randomization *and* pure nondeterminism. The central question is how to find a strategy resolving the pure nondeterminism such that the so-obtained *determinized* program satisfies a given quantitative specification, i.e., bounds on expected outcomes such as the expected final value of a program variable or the probability to terminate in a given set of states. We show how *memoryless and deterministic (MD)* strategies can be obtained in a semi-automatic fashion using deductive verification techniques. For loop-free programs, the MD strategies resulting from our weakest precondition-style framework are correct by construction. This extends to loopy programs, provided the loops are equipped with suitable loop invariants — just like in program verification. We show how our technique relates to the well-studied problem of obtaining strategies in countably infinite Markov decision processes with reachability-reward objectives. Finally, we apply our technique to several case studies.

CCS Concepts: • **Theory of computation** → **Probabilistic computation**; **Logic and verification**; **Invariants**; **Program specifications**; **Pre- and post-conditions**; **Program verification**; **Operational semantics**.

Additional Key Words and Phrases: probabilistic programs, Markov decision processes, strategy synthesis, weakest preexpectations, program verification, quantitative loop invariants

ACM Reference Format:
Kevin Batz, Tom Jannik Biskup, Joost-Pieter Katoen, and Tobias Winkler. 2024. Programmatic Strategy Synthesis: Resolving Nondeterminism in Probabilistic Programs. *Proc. ACM Program. Lang.* 8, POPL, Article 93 (January 2024), 29 pages. https://doi.org/10.1145/3632935

1 INTRODUCTION

Nondeterministic probabilistic programs are like usual imperative programs with the additional abilities to (1) flip coins and (2) choose between multiple execution branches in a *purely nondeterministic* fashion. These programs have various applications: If the nondeterminism models an *uncontrollable adversary*, they can be used to reason about safe abstractions or *underspecifications* of (fully) probabilistic programs, processes, and systems [Kattenbelt et al. 2009, 2010; Kozine and Utkin 2002]. If, on the other hand, the nondeterminism models a *controllable agent*, such programs model games and security mechanisms under stochastic adversaries, and can be used for planning and control, see [Feng et al. 2015; Haesaert et al. 2017] for various applications.

*Batz and Katoen are supported by the ERC AdG 787914 FRAPPANT. Winkler is supported by the DFG RTG 2236 UnRAVeL.

Authors' addresses: Kevin Batz, RWTH Aachen University, Germany, kevin.batz@cs.rwth-aachen.de; Tom Jannik Biskup, RWTH Aachen University, Germany, tom.biskup@rwth-aachen.de; Joost-Pieter Katoen, RWTH Aachen University, Germany, katoen@cs.rwth-aachen.de; Tobias Winkler, RWTH Aachen University, Germany, tobias.winkler@cs.rwth-aachen.de.

ACM 2475-1421/2024/1-ART93
https://doi.org/10.1145/3632935

In many of these settings, a central problem is to determine *strategies* — resolutions of the nondeterminism — satisfying a given specification. In this paper, we consider quantitative specifications imposing bounds on probabilities to terminate in a given set of states or on expected outcomes, e.g., "the expected final value of x is at most 5". We tackle this problem from a programmatic perspective:

Given a nondeterministic probabilistic program C and a quantitative specification, if possible, <u>resolve the nondeterminism in C</u> to obtain a deterministic, but still probabilistic, program C' that satisfies the given quantitative specification.

The idea is that the "determinized" program C' is a *symbolic representation* of a strategy steering the program execution towards satisfying the given quantitative specification.

Modeling and Reasoning with Nondeterministic Probabilistic Programs. To illustrate probabilistic programs with nondeterminism as a means to reason about decision making under stochastic uncertainty, consider the (in)famous *Monty Hall problem*: In a game show, a prize is hidden behind one of three curtains, the other curtains hide nothing. Each of these three curtains hides the prize with equal *probability* 1/3. The game is then played as follows:

(1) The contestant selects one curtain (which is not uncovered yet).
(2) The host uncovers one of the other two curtains *which does not hide the prize.*
(3) Finally, the contestant may —but does not have to— *switch* their choice to the remaining curtain, i.e., the one that is not yet uncovered and not picked initially.

The chosen curtain is then uncovered and the contestant wins whatever is behind it. The question is if switching the curtain in stage (3) increases the chances to win.

The Monty Hall problem is readily modeled as the program in Figure 1 (adapted from [McIver and Morgan 2005]). Variables $cc, pc, hc \in \{1, 2, 3\}$ store the initially chosen contestant curtain, the curtain hiding the prize, and the curtain which is uncovered by the host, respectively. The expression $otherCurtain(c_1, c_2)$ returns the unique curtain c_3 with $c_3 \neq c_1 \wedge c_3 \neq c_2$. Similarly, $rndOtherCurtain(a)$ yields one of the two curtains other than a chosen uniformly at random. The *guarded commands* $\texttt{if}\, \mathsf{true} \rightarrow \{C_1\} \,\square\, \ldots \,\square\, \mathsf{true} \rightarrow \{C_n\}$ model *nondeterministic* choices between the respective branches. To account for the fact that the contestant does not know which curtain hides the prize, the contestant commits to their strategy upfront (lines 1-2).

Strategies for Loop-Free Programs. Using the weakest preexpecation (wp) calculus by McIver and Morgan [2005], one can prove that the *maximal* probability to win is 2/3.

Our technique employs this wp-calculus to construct, in a fully mechanizable way, a strategy in form of <u>strengthenings</u> of the predicates guarding the nondeterministic choices, which attains this winning probability.

More precisely, we obtain a determinized program in which lines 1-2 are replaced by a statement equivalent to $switch$:= true, as *switching in stage (3) is indeed the best strategy*. The choices in lines 3-5 are not resolved as they do not affect the winning probability. In fact, our strategies are generally *permissive* [Dräger et al. 2015]: they do not remove more nondeterminism than necessary.

```
1   if true → {switch := true}
2    □ true → {switch := false} ⨟
3   if true → {cc := 1}
4    □ true → {cc := 2}
5    □ true → {cc := 3} ⨟
6   pc := unif (1, 2, 3) ⨟
7   if pc ≠ cc → {hc := otherCurtain(pc, cc)}
8    □ pc = cc → {hc := rndOtherCurtain(pc)} ⨟
9   if switch → {cc := otherCurtain(cc, hc)}
10   □ ¬switch → {skip}
```

Fig. 1. Monty Hall problem.

Loops, Unbounded Variables, and Invariants. The program from Figure 1 contains no loops and only bounded variables. However, our technique also yields strategies for *possibly unbounded* nondeterministic probabilistic loops, provided they are annotated with suitable *quantitative loop invariants*. To illustrate this, consider the program in Figure 2. It models a variant of the game *Nim*,

```
while $x < N \to \{$
    if $turn = 1 \to \{x := \mathtt{unif}\,(x{+}1,\ x{+}2,\ x{+}3)\}$
    $\square$ $turn = 2 \to \{$
        if true $\to \{x := x + 1\}$
        $\square$ true $\to \{x := x + 2\}$
        $\square$ true $\to \{x := x + 3\}$
    $\}$
    $turn := 3 - turn$
$\}\ \langle I_{Nim} \rangle$
```

Fig. 2. Program modeling the Nim game.

```
while $x < N \to \{$
    if $turn = 1 \to \{x := \mathtt{unif}\,(x{+}1,\ x{+}2,\ x{+}3)\}$
    $\square$ $turn = 2 \to \{$
        if $(x + 1 \equiv_4 N) \vee (x + 2 \equiv_4 N) \to \{x := x + 1\}$
        $\square$ $(x + 1 \equiv_4 N) \vee (x + 3 \equiv_4 N) \to \{x := x + 2\}$
        $\square$ $(x + 1 \equiv_4 N) \vee (x \equiv_4 N) \to \{x := x + 3\}$
    $\}$
    $turn := 3 - turn$
$\}\ \langle I_{Nim} \rangle$
```

Fig. 3. Program representing strategies for Nim .

a 2-player zero-sum game which goes as follows: N tokens are placed on a table. The players take turns; in each turn, the player has to remove 1, 2, or 3 tokens from the table. The first player to remove the last token looses the game. We have annotated the loop with a quantitative invariant I_{Nim} (see Section 7 for details). This invariant certifies that the *maximal* winning probability of the controllable player 2 is at least $2/3$. Now,

> *Given a quantitative specification and suitably strong quantitative loop invariants for all — possibly nested — loops in a program C, our technique automatically yields a program C' where the nondeterministic choices are restricted in such a way that any strategy consistent with C' satisfies the desired quantitative specification.*

The result of applying this technique to our example is given in Figure 3. *Any* strategy playing the game according to this program wins with probability at least $2/3$ *for all values of N*.

Contributions. In summary, this paper makes the following contributions:

Program-level construction of strategies A mechanizable technique to determinize loop-free nondeterministic probabilistic programs in an optimal manner (Theorem 6.3). Given quantitative loop invariants, our technique as well determinizes programs with — possibly multiple nested and sequential — loops, in a mechanizable way (Theorem 6.5).

A novel proof rule for lower bounds on expected outcomes of loops As a by-product of our results, we obtain a generalization of a powerful proof principle for verifying lower bounds on expected outcomes of *deterministic* probabilistic loops to *non*deterministic probabilistic loops — a problem left open by Hark et al. [2020]. See Section 6.2.3 for details.

A bridge to the world of Markov Decision Processes We establish tight connections between our setting and the well-known problem of finding good, or even optimal, strategies for resolving nondeterminism in countably infinite MDPs.

Case studies and examples to demonstrate the applicability of our technique (Section 7).

Limitations and Assumptions. We focus on quantitative specifications imposing lower- or upper bounds on expected outcomes of a program C, i.e., bounds on expected values of random variables w.r.t. the distribution of final states obtained from running C (cf. Section 4.1 for a formal problem statement). We do *not* consider long-run properties such as mean-payoff objectives [Puterman 1994] or general ω-regular properties [Baier and Katoen 2008, Ch. 10]. Further, we restrict to *memoryless and deterministic* strategies (as opposed to history-dependent randomized strategies). For countable state spaces, this is sufficient for (ε-)optimality w.r.t. the considered objectives (cf. Section 2.2.4). For this reason, we assume that program variables range over a countable domain (cf. Section 2.1).

Regarding mechanizability, our approach automatically yields optimal strategies for *loop-free* programs. Hence, if it exists, we can compute a determinized program satisfying the given quantitative specification (cf. Section 4.2 and Theorem 6.3). In the presence of loops, we require that all loops are annotated with sufficiently strong invariants. We do *not* automate the synthesis or the verification of such invariants. Rather, we show that if a nondeterministic program is annotated with such loop invariants, then suitable determinizations can be computed (cf. Section 4.3 and Theorem 6.5).

Paper Structure. Sections 2 and 3 introduce the deductive verification techniques and operational MDP semantics our technique is based on. In Section 4 we provide an informal, illustrative bird's eye view on our approach. Our technique is parameterized by quantitative verification condition generators, which we introduce in Section 5. Program-level synthesis of strategies — the main technical contribution of our paper — is described in Section 6. We provide further case studies in Section 7, discuss related works in Section 8, and conclude in Section 9.

2 NONDETERMINSTIC PROBABILISTIC PROGRAMS

In this section, we introduce the syntax as well as an operational Markov decision process semantics of a simple probabilistic programming language featuring nondeterminstic choices.

2.1 Syntax

Let $\mathsf{Vars} = \{x, y, z, \ldots\}$ be a countably infinite set of *(program) variables* with values[1] from the set $\mathsf{Vals} = \mathbb{Q}_{\geq 0}$. The countably infinite set of *(program) states* is

$$\mathsf{States} \;=\; \{\sigma\colon \mathsf{Vars} \longrightarrow \mathsf{Vals} \;\mid\; \sigma(x) = 0 \text{ for all but finitely many } x \in \mathsf{Vars}\}\,.$$

The set of *predicates* over States is $\mathbb{P} = \big\{\varphi \mid \varphi\colon \mathsf{States} \longrightarrow \{\mathsf{true}, \mathsf{false}\}\big\}$. We write $\sigma \models \varphi$ instead of $\sigma \in \varphi$. A predicate φ is *valid*, denoted $\models \varphi$, if $\sigma \models \varphi$ for every σ, and it is called *unsatisfiable* if $\neg\varphi$ is valid. Programs C in the *probabilistic guarded command language* pGCL adhere to the grammar

$$
\begin{array}{lcll}
C & ::= & \mathtt{skip} & \text{(effectless program)}\\
 & & \mid x := E & \text{(variable assignment)}\\
 & & \mid C \fatsemi C & \text{(sequential composition)}\\
 & & \mid \mathtt{if}\, \varphi_1 \to \{C\} \,\square\, \varphi_2 \to \{C\} & \text{(guarded choice)}\\
 & & \mid \{\, C \,\}\ [p]\ \{\, C \,\} & \text{(probabilistic choice)}\\
 & & \mid \mathtt{while}\, \varphi \to \{C\}\, \langle I \rangle & \text{(loop with invariant annotation } I\text{)}
\end{array}
$$

where E is a function of type $\mathsf{States} \longrightarrow \mathsf{Vals}$ called *expression*, φ_1, φ_2 and φ are predicates from $\mathbb{P}$, and p is a *probability expression* of type $\mathsf{States} \longrightarrow [0, 1]$. For the guarded choice, we require that for every state σ, either $\sigma \models \varphi_1$ or $\sigma \models \varphi_2$ or both hold, i.e., that $\varphi_1 \vee \varphi_2$ is valid. We call a program $C \in$ pGCL *deterministic* if for all $\mathtt{if}\, \varphi_1 \to \{C_1'\} \,\square\, \varphi_2 \to \{C_2'\}$ occurring in C, $\varphi_1 \wedge \varphi_2$ is unsatisfiable. Loops are annotated with *quantitative invariants* I which are functions of type $\mathsf{States} \longrightarrow \mathbb{R}_{\geq 0}^{\infty}$ (see Section 4.3 for details). We often omit the invariant I if it is irrelevant in the current context.

[1]We have chosen the value domain $\mathbb{Q}_{\geq 0}$ for the sake of concreteness. Our results straightforwardly apply to more general (possibly many-sorted) countable domains.

We briefly describe each pGCL construct. `skip` does nothing. $x := E$ evaluates expression E in the current state and assigns the resulting value to variable x. $C_1 \fatsemi C_2$ first executes C_1, and then — if C_1 terminates — C_2. The guarded choice $\mathtt{if}\, \varphi_1 \to \{C_1\} \mathbin{\square} \varphi_2 \to \{C_2\}$ first checks which of the guards φ_1 and φ_2 evaluates to true under the current state. If only one of the guards, say φ_1, evaluates to true, then the guarded choice deterministically executes C_1. If *both* φ_1 and φ_2 evaluate to true, then the guarded choice behaves *nondeterministically* by *either* executing C_1 *or* C_2. Notice that standard conditional choice $\mathtt{if}\ (\,\varphi\,)\ \{\,C_1\,\}\ \mathtt{else}\ \{\,C_2\,\}$ is syntactic sugar for $\mathtt{if}\, \varphi \to \{C_1\} \mathbin{\square} \neg\varphi \to \{C_2\}$. The probabilistic choice $\{\,C_1\,\}\ [p]\ \{\,C_2\,\}$ introduces randomization: In state σ, C_1 is executed with probability $p(\sigma)$ and C_2 is executed with the remaining probability $1 - p(\sigma)$. Finally, the loop $\mathtt{while}\, \varphi \to \{C\}$ executes the loop body C as long as φ evaluates to true, which is the only possible source of non-termination. We conclude this section with an example.

Example 2.1. Consider the program C in Figure 4. Program C is a loop which contains both randomization and nondeterminism. In each iteration, a fair coin is flipped. Depending on the outcome, the loop either terminates or increments x nondeterministically by either 1 or 2. △

2.2 Markov Decision Process Semantics

In this section, we define a formal semantics of pGCL programs in terms of (countably infinite) *Markov decision processes* (*MDP*), based on [Gretz et al. 2012] with adaptions from [Batz et al. 2019].

Formally, an MDP is a quadruple $\mathcal{M} = (S, S_{\mathrm{init}}, A, \mathrm{P})$ where S is a countable non-empty set of states, $S_{\mathrm{init}} \subseteq S$ is a *set* of initial states, A is a finite non-empty set of action labels, and $\mathrm{P}\colon S \times A \times S \longrightarrow [0,1]$ is a transition probability function such that for all $s \in S$ and all $a \in A$, $\sum_{s' \in S} \mathrm{P}(s, a, s') \in \{0, 1\}$. For $s \in S$ we write $A(s) = \{a \in A \mid \sum_{s' \in S} \mathrm{P}(s, a, s') = 1\}$ and require that $|A(s)| \geq 1$ for every s. An MDP $\mathcal{M}$ is called *deterministic*, if $|A(s)| = 1$ for all $s \in S$.

2.2.1 MDP Semantics of pGCL. We first define a small-step execution relation $\longrightarrow$ between *program configurations*. These configurations consist of (i) either a pGCL program C that is still to be executed or a symbol $\Downarrow$ indicating termination and (ii) a program state σ from States. Formally, the countable set Conf of *program configurations* is given by $\mathrm{Conf} \;=\; (\mathrm{pGCL} \cup \{\Downarrow\}) \times \mathrm{States}$. The small-step execution relation is of the form $\longrightarrow \subseteq \mathrm{Conf} \times \{\tau, \alpha, \beta\} \times [0,1] \times \mathrm{Conf}$. Intuitively, we can think of the elements from $\longrightarrow$ as labeled transitions between program configurations. The second component of $\longrightarrow$ is an *action label* $\ell \in \{\tau, \alpha, \beta\}$, and the third component is the transition's probability $q \in [0,1]$. The formal definition of $\longrightarrow$ is standard and given in [Batz et al. 2023a, Appendix B]. In a nutshell, $\longrightarrow$ realizes the intended semantics of pGCL programs as described in Section 2.1. The action label τ is used for all transitions except for those corresponding to a (possibly nondeterministic) guarded command $\mathtt{if}\, \varphi_1 \to \{C_1\} \mathbin{\square} \varphi_2 \to \{C_2\}$. The transition labels α and β distinguish between the branches chosen by such a guarded command. We often write $t \xrightarrow{\ell,q} t'$ instead of $(t, \ell, q, t') \in \longrightarrow$. We also define a binary *successor relation* $t \rightharpoonup t' \iff \exists \ell, q\colon t \xrightarrow{\ell,q} t'$. We say that configuration t' is *reachable* from t if (t, t') is in the reflexive-transitive closure of the relation $\rightharpoonup$. Based on the execution relation $\longrightarrow$ we can now define our MDP semantics:

Definition 2.2. The *operational Markov decision process* $\mathcal{M}(C)$ of program $C \in \mathrm{pGCL}$ is

$$\mathcal{M}(C) \;=\; (S, S_{\mathrm{init}}, A, \mathrm{P}) \text{ , where}$$

(1) $S = \{(C', \sigma') \in \mathrm{Conf} \mid \exists \sigma \in \mathrm{States}\colon (C', \sigma') \text{ is reachable from } (C, \sigma)\}$,
(2) $A = \{\tau, \alpha, \beta\}$ is the set of *action labels*,

```
while c = 0 → {
    {c := 1} [0.5] {
        if true → {x := x + 1}
        □ true → {x := x + 2}
    }
}
```

Fig. 4. Program C from Example 2.1.

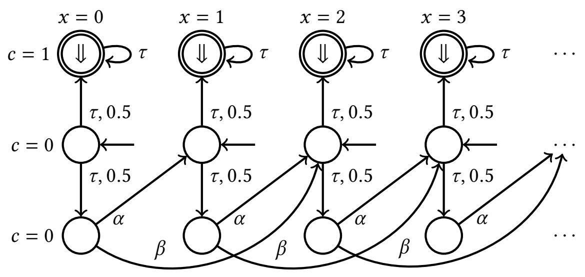

Fig. 5. The semantic MDP $\mathcal{M}(C)$ of program C from Figure 4.

(3) $\mathsf{P}\colon S \times A \times S \longrightarrow [0,1]$ is the *transition probability function* given by

$$\mathsf{P}(t, \ell, t') \;=\; \begin{cases} q & \text{if } t \xrightarrow{\ell, q} t' \\ 0 & \text{else}\,, \end{cases}$$

(4) and $S_{\text{init}} = \{(C, \sigma) \mid \sigma \in \text{States}\}$ are the initial states of $\mathcal{M}(C)$.

Example 2.3. A fragment of the MDP $\mathcal{M}(C)$ of program C from Example 2.1 is sketched in Figure 5, where we assume that $x \in \mathbb{N}$ and $c \in \{0, 1\}$. We depict a fragment reachable from the initial states where $c = 0$ (middle row). Transition probabilities equal to 1 are omitted. △

2.2.2 Strategies. A *strategy* for $\mathcal{M} = (S, S_{\text{init}}, A, \mathsf{P})$ is a function $\mathfrak{S}\colon S^+ \longrightarrow \mathit{Dist}(A)$ satisfying $\mathit{supp}(\mathfrak{S}(s_0 \dots s_n)) \subseteq A(s_n)$ for all $s_0 \dots s_n \in S^+$. Strategy $\mathfrak{S}$ is called *memoryless* if $\mathfrak{S}(s_0 \dots s_n)$ depends only on s_n, and *deterministic* if $\mathfrak{S}(s_0 \dots s_n)$ is a Dirac distribution. Memoryless strategies can be identified with maps of type $S \longrightarrow \mathit{Dist}(A)$, and strategies which are both *memoryless and deterministic* (*MD*) can be identified with maps of type $S \longrightarrow A$. An MD strategy $\mathfrak{S}\colon S \longrightarrow A$ for $\mathcal{M}$ induces a deterministic MDP $\mathcal{M}^{\mathfrak{S}} = (S, S_{\text{init}}, A, \mathsf{P}')$ where for $s, s' \in S$ and $a \in A$,

$$\mathsf{P}'(s, a, s') \;=\; \begin{cases} \mathsf{P}(s, a, s'), & \text{if } a = \mathfrak{S}(s) \\ 0, & \text{otherwise} \end{cases}.$$

We call MDP $\mathcal{M}'$ a *determinization* of $\mathcal{M}$ if there exists an MD strategy $\mathfrak{S}$ such that $\mathcal{M}' = \mathcal{M}^{\mathfrak{S}}$.

2.2.3 Objectives. The fundamental objective considered in this paper is called *reachability-reward*, a mixture of reachability and expected reward objectives. Intuitively, reachability-reward is like standard reachability, with the difference that each state in the target set T is a sink and has a $\mathbb{R}^{\infty}_{\geq 0}$-valued reward that is collected when that state is reached for the first time. The goal is to either minimize or maximize the expected reward. We now formalize reachability-reward. For $T \subseteq S$, the set of finite paths eventually reaching T is

$$\lozenge T \;=\; \{\, s_0 \dots s_m \in S^+ \;\mid\; s_m \in T,\, \forall k \in \{0, \dots, m-1\}\colon s_k \notin T \,\}\,.$$

Moreover, given $s_0 \dots s_m \in S^+$ and strategy $\mathfrak{S}$, we define

$$\mathsf{Prob}_{\mathfrak{S}}(s_0 \dots s_m) \;=\; \prod_{0 \leq k < m} \underbrace{\sum_{\ell \in A} \mathfrak{S}(s_0 \dots s_k)(\ell) \cdot \mathsf{P}(s_k, \ell, s_{k+1})}_{\text{probability to move in one step from } s_k \text{ to } s_{k+1} \text{ under strategy } \mathfrak{S}}\,,$$

with the convention that the empty product (for $m = 0$) is equal to 1. Finally, given $\mathsf{rew}\colon T \longrightarrow \mathbb{R}^{\infty}_{\geq 0}$, we define the function $\mathsf{ExpRew}_{\mathfrak{S}}[\![\mathcal{M}]\!]\,(\mathsf{rew}) : S_{\mathrm{init}} \longrightarrow \mathbb{R}^{\infty}_{\geq 0}$, which maps every initial state s_{init} to its expected (reachability-)reward $\mathsf{ExpRew}_{\mathfrak{S}}[\![\mathcal{M}]\!]\,(\mathsf{rew})\,(s_{\mathrm{init}})$ under strategy $\mathfrak{S}$ by

$$\mathsf{ExpRew}_{\mathfrak{S}}[\![\mathcal{M}]\!]\,(\mathsf{rew})\,(s_{\mathrm{init}}) \;=\; \sum_{s_{\mathrm{init}} s_1 \ldots s_m \in \Diamond T} \mathsf{Prob}_{\mathfrak{S}}\,(s_{\mathrm{init}} s_1 \ldots s_m) \cdot \mathsf{rew}(s_m)\ .$$

Notice that, for constant reward functions $\mathsf{rew}\colon T \longrightarrow \{1\}$, $\mathsf{ExpRew}_{\mathfrak{S}}[\![\mathcal{M}]\!]\,(\mathsf{rew})\,(s_{\mathrm{init}})$ is the probability to reach T from initial state s_{init} under strategy $\mathfrak{S}$. Finally, we define

$$\mathsf{MinExpRew}[\![\mathcal{M}]\!]\,(\mathsf{rew}) \;=\; \inf_{\mathfrak{S} \in \mathit{Strats}} \mathsf{ExpRew}_{\mathfrak{S}}[\![\mathcal{M}]\!]\,(\mathsf{rew}) \qquad (1)$$

$$\text{and} \quad \mathsf{MaxExpRew}[\![\mathcal{M}]\!]\,(\mathsf{rew}) \;=\; \sup_{\mathfrak{S} \in \mathit{Strats}} \mathsf{ExpRew}_{\mathfrak{S}}[\![\mathcal{M}]\!]\,(\mathsf{rew})\ , \qquad (2)$$

where *Strats* denotes the set of all strategies for $\mathcal{M}$ (possibly using memory and/or randomization), and where the infimum and supremum are understood pointwise. If $\mathcal{M}$ is deterministic, then there exists exactly one strategy $\mathfrak{S}$, and thus $\mathsf{MinExpRew}[\![\mathcal{M}]\!]\,(\mathsf{rew}) = \mathsf{MaxExpRew}[\![\mathcal{M}]\!]\,(\mathsf{rew})$. This justifies writing $\mathsf{ExpRew}[\![\mathcal{M}]\!]\,(\mathsf{rew})$ instead of $\mathsf{MinExpRew}[\![\mathcal{M}]\!]\,(\mathsf{rew})$ (or $\mathsf{MaxExpRew}[\![\mathcal{M}]\!]\,(\mathsf{rew})$).

Example 2.4. Reconsider the MDP $\mathcal{M}\,(C)$ from Figure 5. Suppose that reward function rew assigns the value of x to each terminal state in the top row. Then $\mathsf{MaxExpRew}[\![\mathcal{M}\,(C)]\!]\,(\mathsf{rew})\,(C, \sigma) = \sigma(x) + 2$ for every initial state (C, σ) with $\sigma(c) = 0$. This maximal expected reward is attained by the MD strategy that always chooses action β. △

Our ultimate goal is to apply deductive program verification techniques to solve the strategy synthesis problem for MDPs arising from pGCL programs (see also Section 4.1):

Given MDP $\mathcal{M}$, reward function rew, $\sim\ \in \{\leq, \geq\}$, *and thresholds* $\rho\colon S_{\mathrm{init}} \longrightarrow \mathbb{R}^{\infty}_{\geq 0}$,
if it exists, find a *<u>memoryless deterministic</u>* *strategy* $\mathfrak{S}$ *with* $\mathsf{ExpRew}_{\mathfrak{S}}[\![\mathcal{M}]\!]\,(\mathsf{rew}) \sim \rho$
or, equivalently,
if it exists, find a determinization $\mathcal{M}'$ *of* $\mathcal{M}$ *with* $\mathsf{ExpRew}[\![\mathcal{M}']\!]\,(\mathsf{rew}) \sim \rho$.

The comparison relation $\sim$ in the problem statement above refers to the pointwise lifted order $\leq$ (resp. $\geq$) on $\mathbb{R}^{\infty}_{\geq 0}$ to maps of type $S_{\mathrm{init}} \longrightarrow \mathbb{R}^{\infty}_{\geq 0}$. Note that we are interested in finding strategies that guarantee a given threshold for *all* initial states. Such strategies are called *uniform* in the literature.

2.2.4 Existence of Optimal MD Strategies. Strategies that attain the infimum in (1) (resp. the supremum in (2)) are called *optimal.* It is known that *optimal MD strategies always exist in the minimizing setting* [Puterman 1994, Theorem 7.3.6a]. In fact, our theory established in the upcoming sections implies this result for the class of MDPs described by pGCL programs. The above problem is thus guaranteed to have a solution $\mathfrak{S}$ if $\sim$ is $\leq$ and $\rho \geq \mathsf{MinExpRew}[\![\mathcal{M}]\!]\,(\mathsf{rew})$.

The maximizing setting is fundamentally different and more subtle [Blackwell 1967; Ornstein 1969]. First, optimal maximizing strategies do not exist in general, i.e., the supremum in (2) might not be attained. For instance, the MDP $\mathcal{M}$ in Figure 6 (black states only, blue rewards, S_{init} = topmost row) satisfies $\mathsf{MaxExpRew}[\![\mathcal{M}]\!]\,(\mathsf{rew}) = 1$, but for all strategies $\mathfrak{S}$ and initial states $s \in S_{\mathrm{init}}$ we have $\mathsf{ExpRew}_{\mathfrak{S}}[\![\mathcal{M}]\!]\,(\mathsf{rew})\,(s) < 1$. To see this, observe that any strategy that reaches T with positive probability must play some action α (say the n-th one) with positive probability $p > 0$, resulting in an expected reward of at most $p \cdot \frac{n}{n+1} + (1-p)$ which is clearly less than 1. This example shows that, unlike in the minimizing setting, the above problem does *not* necessarily have a solution if $\sim$ is $\geq$ and $\rho \leq \mathsf{MaxExpRew}[\![\mathcal{M}]\!]\,(\mathsf{rew})$, *even if general strategies are allowed.*

On the other hand, there are also situations where MD strategies are strictly less powerful than general strategies. For example, consider the MDP in Figure 6 (black states only, red rewards, S_{init} = topmost row). Here, $\mathsf{MaxExpRew}[\![\mathcal{M}]\!]\,(\mathsf{rew}) = \infty$ as witnessed by the *optimal randomizing strategy*

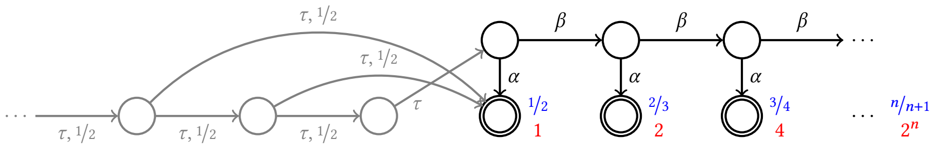

Fig. 6. Example MDP described by Ornstein [1969]. Transition probabilities equal to 1 are omitted.

that plays each action with probability 1/2. In contrast, each MD strategy obviously yields only finite expected reward. MD strategies are still somewhat useful in this example because for every *constant* threshold $\rho \in \mathbb{R}_{\geq 0}$ — no matter how large — we can find an MD strategy that yields expected reward at least ρ for each initial state. However, this does not hold in general. In fact, Ornstein [1969] gives an MDP $\mathcal{M}$ where $\mathsf{MaxExpRew}[\![\mathcal{M}]\!](\mathsf{rew}) = \infty$ which is attained by a randomizing strategy, but for all MD strategies $\mathfrak{S}$ and all $\varepsilon > 0$, there exists $s \in S_{\text{init}}$ such that $\mathsf{ExpRew}_{\mathfrak{S}}[\![\mathcal{M}]\!](\mathsf{rew}) \leq \varepsilon$ (Figure 6, gray & black states, red reward function, S_{init} = gray states).

MD strategies are nonetheless guaranteed to be reasonably powerful under mild assumptions, which we summarize in the following theorem (where (2) is due to [Ornstein 1969]):

Theorem 2.5. *Consider an MDP $\mathcal{M}$ with countable S and reachability-reward function* rew.

(1) *For finite S there exists an MD optimal maximizing strategy [Baier and Katoen 2008, Ch. 10].*
(2) *[Ornstein 1969] If* $\mathsf{MaxExpRew}[\![\mathcal{M}]\!](\mathsf{rew})(s) < \infty$ *for all* $s \in S_{\text{init}}$*, then:*
    (a) $\forall \varepsilon > 0\colon \exists$ *MD strategy* $\mathfrak{S}\colon \mathsf{MaxExpRew}[\![\mathcal{M}]\!](\mathsf{rew}) \geq (1-\varepsilon)\cdot \mathsf{ExpRew}_{\mathfrak{S}}[\![\mathcal{M}]\!](\mathsf{rew})$,
    (b) *If there is an optimal maximizing strategy, then there is an MD optimal maximizing strategy.*

Note that the premise of (2) holds if rew is a bounded function.

3 WEAKEST PREEXPECTATION REASONING

In this section, we introduce the program calculi we use throughout to reason about (minimal and maximal) expected outcomes of nondeterminstic probabilistic programs. Expectation-based reasoning for deterministic probabilistic programs was pioneered by Kozen [Kozen 1983, 1985]. McIver and Morgan [2005] extended expectation-based reasoning to support programs with nondeterminism.

3.1 Expectations

Expectations[2] are the central objects the calculi considered in this paper operate on. They are the quantitative analogue of predicates: Instead of mapping program states to $\{\text{true}, \text{false}\}$, program states are mapped to $\mathbb{R}^{\infty}_{\geq 0} = \mathbb{R}_{\geq 0} \cup \{\infty\}$. Formally, the complete lattice $(\mathbb{E}, \sqsubseteq)$ of *expectations* is

$$\mathbb{E} \;=\; \{ f \mid f\colon \mathsf{States} \to \mathbb{R}^{\infty}_{\geq 0} \} \qquad \text{where} \qquad f \sqsubseteq g \quad \text{iff} \quad \text{for all } \sigma \in \mathsf{States}\colon f(\sigma) \leq g(\sigma)\,.$$

Expectations are denoted by $f, g, \ldots$ and variations thereof. Infima and suprema in this lattice are thus understood pointwise. In particular, pairwise minima and maxima are given by

$$f \sqcap g \;=\; \lambda\sigma.\min\{f(\sigma), g(\sigma)\} \qquad \text{and} \qquad f \sqcup g \;=\; \lambda\sigma.\max\{f(\sigma), g(\sigma)\}\,.$$

Standard arithmetic operations addition $+$ and multiplication $\cdot$ are also understood pointwise, i.e., for $\circ \in \{+, \cdot\}$, we let $f \circ g = \lambda\sigma.f(\sigma) \circ g(\sigma)$, where we set $0 \cdot \infty = \infty \cdot 0 = 0$. The *Iverson bracket*

[2]The terminology is slightly misleading: Expectations can be thought of as random variables on a program's state space rather than an expected value.

C	$\mathsf{dwp}[\![C]\!](f)$	$\mathsf{awp}[\![C]\!](f)$
`skip`	f	f
$x := E$	$f[x \mapsto E]$	$f[x \mapsto E]$
$C_1 \fatsemi C_2$	$\mathsf{dwp}[\![C_1]\!](\mathsf{dwp}[\![C_2]\!](f))$	$\mathsf{awp}[\![C_1]\!](\mathsf{awp}[\![C_2]\!](f))$
`if` $\varphi_1 \to \{C_1\}$ $\square\ \varphi_2 \to \{C_2\}$	$(\varphi_1 \longrightarrow \mathsf{dwp}[\![C_1]\!](f))$ $\sqcap (\varphi_2 \longrightarrow \mathsf{dwp}[\![C_2]\!](f))$	$[\varphi_1] \cdot \mathsf{awp}[\![C_1]\!](f)$ $\sqcup [\varphi_2] \cdot \mathsf{awp}[\![C_2]\!](f)$
$\{C_1\}\ [p]\ \{C_2\}$	$p \cdot \mathsf{dwp}[\![C_1]\!](f) + (1{-}p) \cdot \mathsf{dwp}[\![C_2]\!](f)$	$p \cdot \mathsf{awp}[\![C_1]\!](f) + (1{-}p) \cdot \mathsf{awp}[\![C_2]\!](f)$
`while` $\varphi \to \{C'\}$	$\mathsf{lfp}\, g.\ [\neg\varphi] \cdot f + [\varphi] \cdot \mathsf{dwp}[\![C']\!](g)$	$\mathsf{lfp}\, g.\ [\neg\varphi] \cdot f + [\varphi] \cdot \mathsf{awp}[\![C']\!](g)$

Fig. 7. Inductive definition of $\mathsf{dwp}[\![C]\!](f)$ and $\mathsf{awp}[\![C]\!](f)$ for $f \in \mathbb{E}$. The lfp is taken w.r.t. $(\mathbb{E}, \sqsubseteq)$.

$[\varphi]$ casts a predicate φ into an expectation [Iverson 1962]:

$$[\varphi] \;=\; \lambda\sigma. \begin{cases} 1 & \text{if } \sigma \models \varphi \\ 0 & \text{otherwise}\,. \end{cases}$$

It is convenient to define a *quantitative implication* $\longrightarrow : \mathbb{P} \times \mathbb{E} \longrightarrow \mathbb{E}$ by

$$\varphi \longrightarrow g \;=\; \lambda\sigma. \begin{cases} g(\sigma) & \text{if } \sigma \models \varphi \\ \infty & \text{otherwise}\,. \end{cases}\,.$$

Intuitively, $\longrightarrow$ acts like a filter: if φ evaluates to true, the implication evaluates to the value of the right-hand side's expectation. Otherwise, $\longrightarrow$ evaluates to the top element ∞ of the lattice $(\mathbb{E}, \sqsubseteq)$. We agree on the following order of precedence for the connectives between expectations:

$$\cdot \quad > \quad + \quad > \quad \sqcap \quad > \quad \sqcup \quad > \quad \longrightarrow$$

That is, $\cdot$ takes precedence over $+$, etc. We use brackets to resolve ambiguities. Finally, given $f \in \mathbb{E}$, $x \in \mathsf{Vars}$, and an arithmetic expression E, we define the substitution of x in f by E as

$$f[x \mapsto E] \;=\; \lambda\sigma. f(\sigma[x \mapsto E(\sigma)]), \text{ where for } v \in \mathsf{Vals}, \text{ we set } \sigma[x \mapsto v](y) \;=\; \begin{cases} v, & \text{if } y = x \\ \sigma(y), & \text{o.w.} \end{cases}$$

3.2 Angelic and Demonic Weakest Preexpectations

To reason about minimal and maximal expected outcomes of programs, we introduce two program calculi — *expectation transformers* — which associate to each $C \in \mathsf{pGCL}$ a map of type $\mathbb{E} \longrightarrow \mathbb{E}$.

Definition 3.1 (Weakest preexpectation transformers). Let $C \in \mathsf{pGCL}$ and $f \in \mathbb{E}$. Each of the following is defined by induction on the structure of C in Figure 7:

(1) $\mathsf{dwp}[\![C]\!](f) \in \mathbb{E}$ is the *demonic weakest preexpectation* of C w.r.t. postexpectation f.
(2) $\mathsf{awp}[\![C]\!](f) \in \mathbb{E}$ is the *angelic weakest preexpectation* of C w.r.t. postexpectation f. △

The two transformers differ in the way they interpret nondeterminism: ***d**emonically* vs. ***a**ngelically*. If C is deterministic, then $\mathsf{dwp}[\![C]\!](f)$ and $\mathsf{awp}[\![C]\!](f)$ coincide, in which case we often simply write $\mathsf{wp}[\![C]\!](f)$. Let us briefly go over the individual rules for $\mathcal{T} \in \{\mathsf{dwp}, \mathsf{awp}\}$.

For the effectless program `skip`, $\mathcal{T}[\![\texttt{skip}]\!](f)$ is just f. For assignments $x := E$, we substitute x in f by the assignment's right-hand side E. For sequentially composed programs $C_1 \fatsemi C_2$, we first determine the intermediate preexpectation $\mathcal{T}[\![C_2]\!](f)$, which is then plugged into $\mathcal{T}[\![C_1]\!]$. The — possibly nondeterministic — guarded choice is treated in more detail below. For the probabilistic

```
wp/// 1/2 · (y + 2) + 1/2 · (y + 4)  =  y + 3                this is wp⟦C⟧(x)
 {
     wp/// y + 2                                        this is wp⟦y := y + 2⟧(y)
       y := y + 2
       /// y            y is the postexpectation for this branch obtained below
 }[1/2]{
     wp/// y + 4                                        this is wp⟦y := y + 4⟧(y)
       y := y + 4
       /// y            y is the postexpectation for this branch obtained below
 }⨾
wp/// y                                                     this is wp⟦x := y⟧(x)
 x := y
 /// x                                                   x is the postexpectation
```

reading direction ↑

Fig. 8. Example program with annotations for determining $\mathrm{dwp}[\![C]\!](x)$.

choice $\{ C_1 \}\ [p]\ \{ C_2 \}$, we determine the preexpectations of the two branches and add them up, weighing each branch according to its probability of being executed. Finally, preexpectations of a loop are given by a least fixpoint which, intuitively, is the limit of all finite loop unrollings. We refer to [Kaminski 2019] for an in-depth treatment of expectation-based reasoning.

Now, given $C \in \mathrm{pGCL}$ and $f \in \mathbb{E}$, each $\mathcal{T} \in \{\mathrm{dwp}, \mathrm{awp}\}$ defines an expectation $\mathcal{T}[\![C]\!](f)$, i.e., a map from program states σ to the quantity $\mathcal{T}[\![C]\!](f)(\sigma)$. In what follows, we convey some intuition on this quantity. Assume for the moment that C is deterministic, i.e., C possibly contains randomization but no nondeterminism. In the absence of nondeterminism, $\mathrm{dwp}[\![C]\!](f)$ and $\mathrm{awp}[\![C]\!](f)$ coincide. Thinking of the postexpectation f as a random variable over C's state space, we have

$$\mathrm{wp}[\![C]\!](f)(\sigma) \;=\; \begin{array}{c}\textit{expected value} \text{ of } f \text{ w.r.t. the (sub-)distribution of } \textit{final} \text{ states} \\ \text{reached after executing } C \text{ on } \textit{initial} \text{ state } \sigma.\end{array}$$

The distribution of final states is a *sub*-distribution whenever C does not terminate almost-surely on initial state σ, where the missing probability mass is the probability of divergence. As outlined above, and analogous to Dijkstra's weakest pre*conditions* for standard programs [Dijkstra 1975], $\mathrm{wp}[\![C]\!](f)$ is obtained by recursively applying the rules from Figure 7, i.e., we start with the postexpectation f *at the end of the program* and — as suggested by the rule for sequential composition — move *backwards* through C until we arrive at the beginning, obtaining $\mathrm{wp}[\![C]\!](f)$. This is exemplified in Figure 8, where we annotate[3] the given program for determining $\mathrm{wp}[\![C]\!](x)$. Due to the backward-moving nature of the transformer, these annotations are best read from bottom to top. Complying with the above explanation, the result $\mathrm{wp}[\![C]\!](x) = y + 3$ now tells us that the *expected final value* of x is given by the *initial value* of y plus 3. This is intuitive: As y is not initialized before it is read, the expected final value of x depends on the initial value of y.

Let us now consider a possibly nondeterministic program C. In this case, the final distribution of states obtained from executing C on some initial state σ might not be unique: it generally depends on the resolution of the nondeterminism. It does therefore no longer make sense to speak about *the*

[3]We slightly abuse notation and denote by x the expectation $\lambda\sigma.\sigma(x)$.

```
dwp/// 4 ⊓ 7  =  4
  if true → { dwp/// 4   y := 1   /// y + 3 }
   □ true → { dwp/// 7   y := 4   /// y + 3 } ⨾
dwp/// y + 3
  C
  /// x
```

```
awp/// 4 ⊔ 7  =  7
  if true → { awp/// 4   y := 1   /// y + 3 }
   □ true → { awp/// 7   y := 4   /// y + 3 } ⨾
awp/// y + 3
  C
  /// x
```

Fig. 9. dwp and awp calculations w.r.t. postexpectation x. Here C is the program from Figure 8. Recall that $\mathsf{dwp}[\![C]\!](x) = \mathsf{awp}[\![C]\!](x) = \mathsf{wp}[\![C]\!](x)$ since C is deterministic.

expected value of f w.r.t. the distribution of final states. Instead, we reason about *optimal* values:

$$\mathsf{dwp}/\mathsf{awp}[\![C]\!](f)(\sigma) \;=\; \begin{array}{c}\textit{minimal/maximal} \text{ expected values of } f \text{ w.r.t. } \textit{all} \text{ (sub-)distributions}\\ \text{of } \textit{final} \text{ states reached after executing } C \text{ on } \textit{initial} \text{ state } \sigma\,.\end{array}$$

To see this, consider the preexpectation of a guarded choice in more detail: We have

$$\begin{aligned} &\mathsf{dwp}[\![\mathtt{if}\ \varphi_1 \to \{C_1\} \ \square\ \varphi_2 \to \{C_2\}]\!](f)(\sigma) \;=\; \min\{\mathsf{dwp}[\![C_i]\!](f)(\sigma) \mid \sigma \models \varphi_i\}\\ \text{and}\quad &\mathsf{awp}[\![\mathtt{if}\ \varphi_1 \to \{C_1\} \ \square\ \varphi_2 \to \{C_2\}]\!](f)(\sigma) \;=\; \max\{\mathsf{awp}[\![C_i]\!](f)(\sigma) \mid \sigma \models \varphi_i\}\end{aligned}$$

i.e, if both φ_1 and φ_2 evaluate to true under σ, then the above quantity is the minimum (resp. maximum) of the preexpectations of the two branches C_1 and C_2. Consider the program C' shown in Figure 9 as an example. It contains C from Figure 8 as a subprogram. The annotations on the left-hand side determine $\mathsf{dwp}[\![C']\!](x)$ — mapping initial states to the *minimal* expected final value of x —, and the right-hand side's annotations determine $\mathsf{awp}[\![C']\!](x)$ — mapping initial states to the *maximal* expected final of x. C' first nondeterministically assigns either 1 or 4 to variable y, and then executes C. The minimal expected final value of x is 4, whereas the maximal one is 7.

3.3 MDP Semantics vs. Weakest Preexpectations

There is a tight connection between reachability-reward objectives in MDPs and weakest preexpectations. It is this tight connection which enables us to link our program-level strategy synthesis techniques to the well-known problem of synthesizing strategies in MDPs. More concretely, $\mathsf{dwp}[\![C]\!](f)(\sigma)$ and $\mathsf{awp}[\![C]\!](f)(\sigma)$ are the minimal and maximal *expected rewards* in the MDP $\mathcal{M}(C)$, respectively, where postexpectation f induces the reward function: upon reaching a terminal configuration $(\Downarrow, \sigma')$, a reward of $f(\sigma')$ is collected. Formally:

Theorem 3.2 ([Batz et al. 2019; Gretz et al. 2012]). *Let* $C \in \mathsf{pGCL}$, $f \in \mathbb{E}$ *a post-expectation. Moreover, let* $\mathsf{rew}_f \colon \{\Downarrow\} \times \mathsf{States} \longrightarrow \mathbb{R}_{\geq 0}^{\infty}$ *be* $\mathsf{rew}_f(\Downarrow, \sigma') = f(\sigma')$. *Then, for all* $\sigma \in \mathsf{States}$,

$$\begin{aligned}\mathsf{dwp}[\![C]\!](f)(\sigma) &= \mathsf{MinExpRew}[\![\mathcal{M}(C)]\!]\left(\mathsf{rew}_f\right)(C,\sigma), \textit{ and}\\ \mathsf{awp}[\![C]\!](f)(\sigma) &= \mathsf{MaxExpRew}[\![\mathcal{M}(C)]\!]\left(\mathsf{rew}_f\right)(C,\sigma)\,.\end{aligned}$$

In particular, if C *is deterministic, then so is* $\mathcal{M}(C)$, *and we have*

$$\mathsf{wp}[\![C]\!](f)(\sigma) \;=\; \mathsf{ExpRew}[\![\mathcal{M}(C)]\!]\left(\mathsf{rew}_f\right)(C,\sigma)\,.$$

The dwp and awp calculi can thus be understood as a means to reason deductively about reachability-reward objectives of possibly infinite-state MDPs modeled by pGCL programs.

$$\frac{C_1' \multimap C_1 \qquad C_2' \multimap C_2}{C_1' \fatsemi C_2' \multimap C_1 \fatsemi C_2} \qquad \frac{\varphi_1' \models \varphi_1 \qquad \varphi_2' \models \varphi_2 \qquad \models \varphi_1' \vee \varphi_2' \qquad C_1' \multimap C_1 \qquad C_2' \multimap C_2}{\mathtt{if}\, \varphi_1' \to \{C_1'\} \,\square\, \varphi_2' \to \{C_2'\} \multimap \mathtt{if}\, \varphi_1 \to \{C_1\} \,\square\, \varphi_2 \to \{C_2\}}$$

$$\frac{C_1' \multimap C_1 \qquad C_2' \multimap C_2}{\{\, C_1' \,\}\ [p]\ \{\, C_2' \,\} \multimap \{\, C_1 \,\}\ [p]\ \{\, C_2 \,\}} \qquad \frac{C' \multimap C}{\mathtt{while}\, \varphi \to \{C'\} \multimap \mathtt{while}\, \varphi \to \{C\}}$$

Fig. 10. Rules defining the implements relation $\multimap$. Here $\models$ denotes *entailment* between predicates, i.e., $\varphi \models \varphi'$ if for all states σ it holds that $\sigma \models \varphi$ implies $\sigma \models \varphi'$.

4 A BIRD'S EYE VIEW: PROGRAMMATIC STRATEGY SYNTHESIS

Before we deal with the fully fledged formalization of our approach, we set the stage in this section by (i) formalizing our problem statement (Section 4.1), (ii) giving an informal description of our approach for *loop-free* programs (Section 4.2), and (iii) demonstrating how these technique generalize to programs *with* loops in Section 4.3. All of this is done in an example-driven manner.

4.1 Problem Statement

Recall the synthesis problem for MDPs $\mathcal{M}$ from Section 2.2.3 of finding determinizations $\mathcal{M}'$ which guarantee bounds on the expected rewards of interest. If the state space of $\mathcal{M}$ is finite, it is known that the problem can be solved in polynomial time via linear programming [Puterman 1994]. However, this technique is in general not applicable to countably *infinite-state* MDPs, which arise naturally from pGCL programs. Our goal is to obtain *program-level strategy synthesis techniques* by means of weakest preexpectation reasoning. Towards lifting the synthesis problem to programs, we formalize the notions of *implementations* and *determinizations.*

Definition 4.1. The *implementation* relation $\multimap \ \subseteq \mathsf{pGCL} \times \mathsf{pGCL}$ is the smallest partial order on pGCL satisfying the rules given in Figure 10. If $C' \multimap C$, then we say that C' *implements* C. If moreover C' is deterministic, then we say that C' is a *determinization* of C. △

If $C' \multimap C$, then C' and C coincide syntactically up to the guards occurring in the guarded choices. The guards in C' may be *strengthened* to resolve some of the nondeterministic choices from C, which is formalized by the premises $\varphi_1' \models \varphi_1$ and $\varphi_2' \models \varphi_2$ in the rule for guarded choices. The premise $\models \varphi_1' \vee \varphi_2'$ ensures that C' does not eliminate *all* choices from some guarded choice in C.

Example 4.2. Consider the programs C_1 (left) C_2 (middle), and C (right).

$\mathtt{if}\, y \leq z \to \{x := y\}$	$\mathtt{if}\, y \leq z \to \{x := y\}$	$\mathtt{if}\, \mathsf{true} \to \{x := y\}$
$\square\ y > z \to \{x := z\} \fatsemi$	$\square\ y \geq z \to \{x := z\} \fatsemi$	$\square\ \mathsf{true} \to \{x := z\} \fatsemi$
$\{x := 0\}\ [1/2]\ \{x := 2 \cdot x\}$	$\{x := 0\}\ [1/2]\ \{x := 2 \cdot x\}$	$\{x := 0\}\ [1/2]\ \{x := 2 \cdot x\}$

We have $C_1 \multimap C_2 \multimap C$. The strengthened guards $y \leq z$ and $y \geq z$ in C_2 resolve some of the nondeterminism from C. Program C_2 is, however, *not* a determinization of C since C_2 behaves nondeterministically whenever $z = y$ holds initially. Program C_1, on the other hand, *is* a determinization of both C_2 and C since the guards $y \leq z$ and $y > z$ are mutually exclusive. △

Since $C' \multimap C$ possibly permits less nondeterministic choices than C, it is an immediate, yet important, characteristic of C' that minimal (resp. maximal) expected outcomes of C' only get larger (resp. smaller) when compared to expected outcomes of C. Formally:

Lemma 4.3. *Let $C, C' \in \mathsf{pGCL}$ and $f \in \mathbb{E}$. We have*

$$C' \multimap C \qquad \text{implies} \qquad \mathsf{dwp}[\![C]\!](f) \sqsubseteq \mathsf{dwp}[\![C']\!](f) \quad \text{and} \quad \mathsf{awp}[\![C]\!](f) \sqsupseteq \mathsf{awp}[\![C']\!](f)\,.$$

Proof. See [Batz et al. 2023a, Appendix A]. □

With the notions of determinizations at hand, we formalize our problem statement:

$$\boxed{\begin{array}{c}\textit{Given } C \in \mathrm{pGCL}, \textit{ postexpectation } f \in \mathbb{E}, \ \sim\ \in \{\sqsubseteq, \sqsupseteq\}, \textit{ and a threshold } g \in \mathbb{E}, \\ \textit{if it exists, find a } \text{determinization } C' \textit{ of } C \textit{ with } \mathrm{wp}[\![C']\!](f) \sim g.\end{array}}$$

Due to the tight connection between weakest preexpectations and reachability-rewards in MDPs (Theorem 3.2), the above synthesis problem can be understood as a synthesis problem for *MDPs*: Each program C induces a countable MDP $\mathcal{M}(C)$. If C' is a determinization of C, then $\mathcal{M}(C')$ is a determinization of $\mathcal{M}(C)$. Hence, from an MDP perspective, our problem statement reads:

$$\boxed{\begin{array}{c}\textit{Given } C \in \mathrm{pGCL}, \textit{ postexpectation } f \in \mathbb{E}, \ \sim\ \in \{\leq, \geq\}, \textit{ and a threshold } g \in \mathbb{E}, \\ \textit{if it exists, find } C' \textit{ such that } \mathcal{M}(C') \textit{ is a determinization of } \mathcal{M}(C) \\ \textit{and for all } \sigma \in \mathsf{States}\colon \mathsf{ExpRew}[\![\mathcal{M}(C')]\!]\left(\mathsf{rew}_f\right)(C', \sigma) \sim g(\sigma).\end{array}}$$

4.2 First Step: Optimal Determinizations for Loop-Free Programs

Our first key insight is that, for *loop-free* programs C, we can compute[4] — in a purely syntactic manner — *optimal* determinizations of C. Put more formally, given loop-free C and postexpectation f, there are effectively constructible determinizations $C_{\min}$ and $C_{\max}$ of C with

$$\mathsf{dwp}[\![C]\!](f) \;=\; \mathsf{wp}[\![C_{\min}]\!](f) \qquad \text{and} \qquad \mathsf{awp}[\![C]\!](f) \;=\; \mathsf{wp}[\![C_{\max}]\!](f)\ .$$

We exemplify the construction of $C_{\min}$. The construction for $C_{\max}$ is dual.

Example 4.4. Reconsider the nondeterministic program C from Example 4.2 and fix the postexpectation $f := x$. Below we give annotations for determining $\mathsf{dwp}[\![C]\!](x)$ — the expectation which maps every initial state to the minimal expected final value of x:

```
$^{\mathsf{dwp}}/\!\!/\!\!/\ y \sqcap z$
  $\mathtt{if}\ \mathtt{true} \rightarrow \{\ ^{\mathsf{dwp}}/\!\!/\!\!/\ y \quad x := y \quad /\!\!/\!\!/\ x\}$
   $\Box\ \mathtt{true} \rightarrow \{\ ^{\mathsf{dwp}}/\!\!/\!\!/\ z \quad x := z \quad /\!\!/\!\!/\ x\}\,⨾$
$^{\mathsf{dwp}}/\!\!/\!\!/\ 1/2 \cdot 0 + 1/2 \cdot 2 \cdot x \;=\; x$
  $\{\ ^{\mathsf{dwp}}/\!\!/\!\!/\ 0 \quad x := 0 \quad /\!\!/\!\!/\ x\}\ [1/2]\ \{\ ^{\mathsf{dwp}}/\!\!/\!\!/\ 2 \cdot x \quad x := 2 \cdot x \quad /\!\!/\!\!/\ x\}$
  $/\!\!/\!\!/\ x$
```

Since C is loop-free, these annotations are obtained in a syntactic manner by recursively applying the rules given in Figure 7. The topmost annotation tells us that $\mathsf{dwp}[\![C]\!](x) = y \sqcap z$, i.e., the minimal expected *final* value of x is the minimum of the *initial* values of y and z. Towards constructing $C_{\min}$, consider the highlighted intermediate preexpectations y and z. These annotations tell us that the expected final value of x will be y *if C executes the <u>first</u> branch*, and that it will be z *if C executes the <u>second</u> branch*. Hence, we can readily read off strengthenings of the guards to resolve the nondeterminism in an optimal way: If $y < z$ holds, the first branch should be taken. Conversely, if $y > z$ holds, the second branch should be taken. In case $y = z$ holds, both choices are optimal. We can thus construct the following implementation C' of C:

$$\mathtt{if}\ y \leq z \rightarrow \{x := y\} \ \Box\ y \geq z \rightarrow \{x := z\}\,⨾\, \{\, x := 0\,\}\ [1/2]\ \{\, x := 2 \cdot x\,\}$$

Program C' is still nondeterministic if $y = z$ holds initially. This nondeterminism can, however, be resolved arbitrarily in the sense that *any* determinization of C' will be optimal. We call C' an

[4]Assuming that expectations are represented syntactically in some sufficiently expressive formal language, e.g., the one from [Batz et al. 2021b] extended by $\sqcap$- and $\sqcup$ operators.

optimal permissive determinization of C w.r.t. postexpectation x. C' is now easily determinized by, e.g., turning one of the inequalities, say $y \leq z$, into a strict one, obtaining (one choice for) $C_{\min}$:

$$\texttt{if}\, y < z \rightarrow \{x := y\} \,\square\, y \geq z \rightarrow \{x := z\} \,\text{⨾}\, \{\, x := 0 \,\}\ [1/2]\ \{\, x := 2 \cdot x \,\}$$

The construction for awp and $C_{\max}$ is dual by flipping the inequalities. △

We formalize our construction of such optimal (permissive) determinizations for arbitrary *loop-free* programs in Section 6. Now reconsider our problem statement from Section 4.1:

Given $C \in$ pGCL, postexpectation f, $\sim\ \in \{\sqsubseteq, \sqsupseteq\}$, and a threshold $g \in \mathbb{E}$,
if it exists, find a determinization C' *of C with* wp$[\![C']\!](f) \sim g$.

In Section 6 further below, we show that the solution for loop-free C is as follows:

- If $\sim$ is $\sqsubseteq$, then C' exists if and only if dwp$[\![C]\!](f) \sqsubseteq g$, in which case C' is given by $C_{\min}$.
- If $\sim$ is $\sqsupseteq$, then C' exists if and only if awp$[\![C]\!](f) \sqsupseteq g$, in which case C' is given by $C_{\max}$.

That is, C' exists iff the minimal (resp. maximal) expected final value of f is upper (resp. lower) bounded by $g(\sigma)$ for every initial state σ. Moreover, C' can be constructed as exemplified above.

Remark 4.5. Our results do not imply decidability of the existence of the sought-after determinizations. Depending on the arithmetic necessary for expressing f, dwp$[\![C]\!](f)$, awp$[\![C]\!](f)$, or g, quantitative entailments of the form dwp$[\![C]\!](f) \sqsubseteq g$ or awp$[\![C]\!](f) \sqsupseteq g$ are often undecidable. △

4.3 Second Step: From Quantitative Loop Invariants to Determinizations of Loops

Naturally, constructing determinizations of loops is more involved. Reasoning about minimal and maximal expected outcomes of loops requires reasoning about least fixpoints (cf. Figure 7), which are uncomputable in general [Kaminski et al. 2019]. How can we nonetheless determinize loops as asked for by our problem statement? Our primary observation is that

quantitative loop invariants yield determinizations of nondeterministic probabilistic loops .

These quantitative loop invariants are generally hard to find, let alone algorithmically. Our key insight here is that once we have a quantitative loop invariant at hand, we can use it to find determinizations of loops. Our approach thus applies to any technique which verifies pGCL programs by means of quantitative loop invariant-based reasoning satisfying the assumptions formalized in Section 6. In what follows, we first introduce quantitative loop invariants, and then present an example of our construction for obtaining determinizations guided by these invariants.

4.3.1 *Quantitative Loop Invariants.*

Analogous to classical Floyd-Hoare logic for standard programs, quantitative loop invariants establish bounds on preexpectations of a loop by reasoning inductively about *one* arbitrary, but fixed, loop iteration. We introduce the following notions [Kaminski 2019]:

Definition 4.6. Let $C = \texttt{while}\,\varphi \rightarrow \{C_{\text{body}}\}$, $I, f \in \mathbb{E}$, and $\mathcal{T} \in \{\text{dwp}, \text{awp}\}$.

(1) If $[\varphi] \cdot \mathcal{T}[\![C_{\text{body}}]\!](I) + [\neg\varphi] \cdot f \sqsubseteq I$, then we call I a $\mathcal{T}$-*super*invariant of C (w.r.t. f).
(2) If $[\varphi] \cdot \mathcal{T}[\![C_{\text{body}}]\!](I) + [\neg\varphi] \cdot f \sqsupseteq I$, then we call I a $\mathcal{T}$-*sub*invariant of C (w.r.t. f). △

Hence, to determine whether I is a super/sub-invariant of C, it suffices to determine $\mathcal{T}[\![C_{\text{body}}]\!](I)$ — the preexpectation of the loop *body* — and to check whether the respective inequality holds. By *Park induction*, every $\mathcal{T}$-*super*invariant I of C w.r.t. f upper-bounds $\mathcal{T}[\![\texttt{while}\,\varphi \rightarrow \{C_{\text{body}}\}]\!](f)$.

Theorem 4.7 ([Park 1969]). *Let $C = \texttt{while}\,\varphi \rightarrow \{C_{\text{body}}\}$, let $I, f \in \mathbb{E}$, and $\mathcal{T} \in \{\text{dwp}, \text{awp}\}$. If I is a $\mathcal{T}$-superinvariant of C w.r.t. f, then $\mathcal{T}[\![C]\!](f) \sqsubseteq I$.*

Example 4.8 (Upper bounds from superinvariants). Consider the loop

$$C \quad = \quad \texttt{while}\, c \neq 0 \rightarrow \underbrace{\{\{\, c := 0 \,\}\ [1/2]\ \{\, \texttt{if}\, \text{true} \rightarrow \{x := x^2\} \,\square\, \text{true} \rightarrow \{x := x + 1\}\,\}\}}_{=:C_{\text{body}}} .$$

On every iteration, C flips a fair coin and, depending on the outcome, either terminates by setting c to 0, or continues iterating after nondeterministically squaring x or incrementing x by one.

Now suppose we wish to upper-bound $\mathsf{dwp}[\![C]\!](x)$ — the expectation which maps every initial state to the minimal expected final value of x — by $[c \neq 0] \cdot (x + 1) + [c = 0] \cdot x =: I$. In words, if initially $c \neq 0$ holds, then the minimal expected final value is upper-bounded by the initial value of x plus 1. Conversely, if initially $c = 0$ holds, then x is not modified at all. We employ Theorem 4.7 to prove that $\mathsf{dwp}[\![C]\!](x) \sqsubseteq I$ indeed holds[5] by verifying that I is a dwp-superinvariant of C w.r.t. x:

$$\begin{aligned} [c \neq 0] \cdot \mathsf{dwp}[\![C_{\text{body}}]\!](I) + [c = 0] \cdot x \;&=\; [c \neq 0] \cdot \big((x + 1) \sqcap 1/2 \cdot (x^2 + x + 1)\big) + [c = 0] \cdot x \\ &\sqsubseteq\; [c \neq 0] \cdot (x + 1) + [c = 0] \cdot x \;=\; I\,. \end{aligned} \qquad \triangle$$

Obtaining *lower* bounds from *sub*invariants requires additional conditions, see Section 4.3.3.

4.3.2 From Loop Invariants to Loop Determinization. How can we use the concept of superinvariants for finding determinizations as demanded by our problem statement? Let us, for the moment, focus on the conceptually simpler case of aiming for determinizations which guarantee *upper* bounds on expected outcomes, i.e., let C be a loop and $f, g \in \mathbb{E}$ and suppose we want to find a determinization C' of C with $\mathsf{wp}[\![C']\!](f) \sqsubseteq g$. We proceed as follows:

(1) Find a dwp-superinvariant I of C w.r.t. f such that I satisfies $I \sqsubseteq g$, and
(2) *compute* — guided by I — a determinization C' of C such that I *as well* is a dwp-superinvariant of C' w.r.t. f, i.e., determinizing C to C' *preserves superinvariance* of I w.r.t. f.

Step (1) is undecidable in general. We assume that I is provided by some external means. Step (2), on the other hand, is as *syntactic* as our construction for loop-free programs from Section 4.2. Steps (1) and (2) together solve the given problem instance because

$$\mathsf{wp}[\![C']\!](f) \quad \overset{\text{Thm. 4.7}}{\sqsubseteq} \quad I \quad \overset{\text{assumption}}{\sqsubseteq} \quad g\,.$$

Example 4.9. Reconsider the loop C from Example 4.8 with postexpectation x. Now fix expectation $x + 1 =: g$ and suppose we wish to find a determinization C' of C with $\mathsf{wp}[\![C']\!](x) \sqsubseteq g$, i.e., the expected final value of x shall be no greater than one plus the initial value of x for all initial states. Recall from Example 4.8 that expectation $I = [c \neq 0] \cdot (x + 1) + [c = 0] \cdot x$ is a dwp-superinvariant of C w.r.t. x. Moreover, we have $I \sqsubseteq g$. Hence, if we can determinize C to C' in such a way that I remains a superinvariant of C' w.r.t. x, we are done.

For that, recall from Section 4.2 that we can determinize *loop-free* programs in an optimal permissive manner. In particular, for the loop-free body C_{body} of C we can construct C'_{body} such that

$$\mathsf{dwp}[\![C_{\text{body}}]\!](I) \quad = \quad \mathsf{wp}[\![C'_{\text{body}}]\!](I)\,.$$

Consequently, $C' = \mathtt{while}\, c \neq 0 \to \{C'_{\text{body}}\}$ will be a determinization of C satisfying

$$[c \neq 0] \cdot \mathsf{dwp}[\![C_{\text{body}}]\!](I) + [c = 0] \cdot x \overset{\text{by construction}}{=} [c \neq 0] \cdot \mathsf{wp}[\![C'_{\text{body}}]\!](I) + [c = 0] \cdot x \overset{\text{see Ex. 4.8}}{\sqsubseteq} I$$

which implies

$$\mathsf{wp}[\![C']\!](x) \quad \overset{\text{Thm. 4.7}}{\sqsubseteq} \quad I \qquad \text{and therefore, by transitivity, also} \qquad \mathsf{wp}[\![C']\!](x) \quad \sqsubseteq \quad g\,.$$

[5]This can be proven automatically using the deductive verifier Caesar [Schröer et al. 2023], see [Batz et al. 2023a, Appendix D].

To construct C'_{body}, we proceed as in Section 4.2: Annotate C_{body} for determining $\mathrm{dwp}[\![C_{\text{body}}]\!](I)$ and derive the strengthenings for the guards from the highlighted intermediate preexpectations:

```
    ${}^{\mathrm{dwp}}/\!\!/\!\!/\ \ldots$ (not relevant)
     { ${}^{\mathrm{dwp}}/\!\!/\!\!/\ \ldots$ (not relevant)   $c := 0$   $/\!\!/\!\!/\ I$}
     [1/2]
     {    ${}^{\mathrm{dwp}}/\!\!/\!\!/\ \ldots$ (not relevant)
          if true → { ${}^{\mathrm{dwp}}/\!\!/\!\!/\ [c \neq 0] \cdot (x \cdot x + 1) + [c = 0] \cdot x \cdot x$   $x := x \cdot x$   $/\!\!/\!\!/\ I$}
          □ true → { ${}^{\mathrm{dwp}}/\!\!/\!\!/\ [c \neq 0] \cdot (x + 2) + [c = 0] \cdot (x + 1)$   $x := x + 1$   $/\!\!/\!\!/\ I$}
     }
     $/\!\!/\!\!/\ I := [c \neq 0] \cdot (x + 1) + [c = 0] \cdot x$
```

Thus, whenever $I[x \mapsto x \cdot x](\sigma) < I[x \mapsto x + 1](\sigma)$ holds in the current variable valuation σ, the first branch must be taken. Conversely, if $I[x \mapsto x \cdot x](\sigma) > I[x \mapsto x + 1](\sigma)$ holds, the second branch must be taken. Finally, in case $I[x \mapsto x \cdot x](\sigma) = I[x \mapsto x + 1](\sigma)$ holds, we may freely choose between the two branches. Now, for fixed state σ and recalling that $\sigma(x) \geq 0$, we have

$$I[x \mapsto x \cdot x](\sigma) \leq I[x \mapsto x + 1](\sigma) \qquad \text{iff} \qquad \sigma(x)^2 \leq \sigma(x) + 1 \qquad \text{iff} \qquad \sigma(x) \leq 1/2 \cdot (1 + \sqrt{5})\,,$$

which yields the *correct-by-construction* determinization C' of C satisfying $\mathrm{wp}[\![C']\!](x) \sqsubseteq g = x + 1$:

```
while $c \neq 0$ → {$c := 0$
                    [1/2]
                   if $x \leq 1/2 \cdot (1 + \sqrt{5})$ → {$x := x \cdot x$} □ $x > 1/2 \cdot (1 + \sqrt{5})$ → {$x := x + 1$}} .        △
```

4.3.3 Subinvariants and Lower Bounds. We have so far disregarded the case where we want to find determinizations which guarantee *lower* bounds on expected outcomes, i.e., for loop C and $f, g, \in \mathbb{E}$, find a determinization C' of C with $g \sqsubseteq \mathrm{wp}[\![C']\!](f)$. Compared to upper-bound reasoning, lower-bounding preexpectations of loops is more involved. $\mathcal{T}$-*sub*invariants I of C do *not* necessarily lower-bound $\mathcal{T}[\![C]\!](f)$. This is not surprising as this is already the case for standard weakest pre*conditions* for non-probabilistic programs, which are subsumed by weakest pre*expectations*. In the realm of weakest preconditions, expectations are replaced by predicates and $\sqsubseteq$ corresponds to $\models$, i.e., entailment between predicates. Proving $\varphi \models \mathrm{wp}[\![C]\!](\psi)$ corresponds to establishing a *total* correctness property of C, which generally requires additional *side conditions* such as termination. However, if we assume that we have such side conditions at hand so that

$$\underbrace{I \text{ is } \mathcal{T}\text{-subinvariant of } C \text{ w.r.t. } f \ \wedge\ \textit{side conditions}}_{\text{denote the validity of these verification conditions by } \mathrm{vc}[\![C]\!](f) \,\in\, \{\text{true}, \text{false}\}} \qquad \text{implies} \qquad \mathcal{T}[\![C]\!](f) \ \sqsupseteq\ I\,,$$

we may proceed in a way analogous to the above procedure for upper bounds:

(1) Find an awp-*sub*invariant I such that both $\mathrm{vc}[\![C]\!](f)$ and $I \sqsupseteq g$ hold, and
(2) *compute* — guided by I — a determinization C' of C such that $\mathrm{vc}[\![C']\!](f)$ is satisfied *as well*, i.e., determinizing C into C' *preserves validity of the verification conditions.*

These *side conditions* come in different flavors[6] and their restrictiveness typically depends on the expressive power of I and the postexpectation f. Moreover, finding such side conditions is an active field of research [Hark et al. 2020]. This motivates our general framework presented in Section 6, where we abstract from the specific side conditions under consideration. This framework may then be instantiated with verification conditions tailored to the specific problem instances under

[6]For instance, if both I and f are bounded by 1, it suffices to show that C is *demonically almost-surely terminating* (cf. Theorem 6.9). If I or f are unbounded, more restrictions are needed [Hark et al. 2020].

C	$\mathsf{vc}^{\mathcal{T}}_{\mathfrak{C}}[\![C]\!](f)$
`skip`	true
$x := E$	true
$C_1 \fatsemi C_2$	$\mathsf{vc}^{\mathcal{T}}_{\mathfrak{C}}[\![C_1]\!](\mathcal{T}^*[\![C_2]\!](f)) \wedge \mathsf{vc}^{\mathcal{T}}_{\mathfrak{C}}[\![C_2]\!](f)$
$\mathtt{if}\, \varphi_1 \to \{C_1\} \,\square\, \varphi_2 \to \{C_2\}$	$\mathsf{vc}^{\mathcal{T}}_{\mathfrak{C}}[\![C_1]\!](f) \wedge \mathsf{vc}^{\mathcal{T}}_{\mathfrak{C}}[\![C_2]\!](f)$
$\{ C_1 \}\ [p]\ \{ C_2 \}$	$\mathsf{vc}^{\mathcal{T}}_{\mathfrak{C}}[\![C_1]\!](f) \wedge \mathsf{vc}^{\mathcal{T}}_{\mathfrak{C}}[\![C_2]\!](f)$
$\mathtt{while}\, \varphi \to \{C'\}\, \langle I \rangle$	$\mathfrak{C}(C, f) \wedge \mathsf{vc}^{\mathcal{T}}_{\mathfrak{C}}[\![C']\!](I)$

Fig. 11. Inductive definition of the verification condition of C w.r.t. f, $\mathfrak{C}$, and $\mathcal{T}$.

consideration. Finally, we remark that our framework also handles determinizations of *arbitrary* pGCL programs — possibly containing nested- and sequenced loops.

4.4 On Completeness and Inherent Incompleteness of our Approach

We show in Section 6.1 that our approach for upper bounds as outlined in Section 4.3.2 is *complete*: For all pGCL programs C and all postexpectations f, our framework can (in theory) produce a determinization $C' \multimap C$ such that $\mathsf{wp}[\![C']\!](f) = \mathsf{dwp}[\![C]\!](f)$. In words, we can find a determinization that actually realizes the *minimal* preexpectation. This matches a known result about countably infinite MDP, namely that MD strategies for minimizing expected rewards independent of the initial state always exist (cf. Section 2.2.4). The situation is fundamentally different for the maximizing expectation transformer awp: For some C and f there does *not* exist a deterministic $C' \multimap C$ such that $\mathsf{wp}[\![C']\!](f) = \mathsf{awp}[\![C]\!](f)$. A simple counterexample is the (non-probabilistic) loop

$$C \quad = \quad \mathtt{while}\, c \neq 0 \to \{\mathtt{if}\, \mathsf{true} \to \{n := n+1\} \,\square\, \mathsf{true} \to \{c := 0\}\}$$

with postexpectation $f = {}^{n}/_{n+1}$. Assuming that $n \in \mathbb{N}$, this program realizes the MDP in Figure 6 (black states & blue reward function). It is clear that $\mathsf{awp}[\![C]\!](f) = [c \neq 0] \cdot 1 + [c = 0] \cdot {}^{n}/_{n+1}$, but no determinization C' of C can actually terminate in a state where $n = 1$. However, recall that under the mild assumptions of Theorem 2.5, existence of (ϵ-)optimal determinizations is guaranteed.

5 REASONING WITH VERIFICATION CONDITIONS

Towards our framework for determinizing pGCL programs, this section introduces a simple verification condition generator adapted from [Navarro and Olmedo 2022]. A distinguishing aspect of our formulation is that — following the motivation from the previous section — our verification condition generator is parameterized by the invariant-based proof rule for reasoning about loops.

Auxiliary Expectation Transformers. To simplify notation for reasoning about (possibly nested- or sequenced) loops, recall from Section 2.1 that all loops occurring in a pGCL program C are annotated with a quantitative loop invariant $I \in \mathbb{E}$, i.e., all loops in C are of the form $\mathtt{while}\, \varphi \to \{C'\}\, \langle I \rangle$. We define for each $\mathcal{T} \in \{\mathsf{dwp}, \mathsf{awp}\}$ (and $\mathcal{T} = \mathsf{wp}$ in case we deal with deterministic programs) an auxiliary expectation transformer $\mathcal{T}^*[\![C]\!]$. The inductive definition of $\mathcal{T}^*[\![C]\!](f)$ is completely analogous to the definition of $\mathcal{T}[\![C]\!](f)$, *except* for loops, for which we define

$$\mathcal{T}^*[\![\mathtt{while}\, \varphi \to \{C'\}\, \langle I \rangle\,]\!](f) \;\coloneqq\; I\,.$$

In particular, for loop-free C, $\mathcal{T}^*[\![C]\!]$ and $\mathcal{T}[\![C]\!]$ coincide. As is standard in verification condition-based program verification [Leino 2010], $\mathcal{T}^*[\![C]\!]$ replaces the (generally uncomputable) least fixpoint by the (externally provided) annotated invariant so that determining $\mathcal{T}^*[\![C]\!](f)$ reduces to syntactic reasoning. The idea is then to define suitable *verification conditions* for C and f in such a way that the validity of these conditions implies that $\mathcal{T}^*[\![C]\!](f)$ upper- or lower-bounds $\mathcal{T}[\![C]\!](f)$.

The Parametric Verification Condition Generator. In order to parameterize our verification condition generator by the invariant-based proof rule employed for approximating expected outcomes of loops, we consider objects $\mathfrak{C}$ of type $\{C \in \mathsf{pGCL} \mid C = \mathtt{while}\,\varphi \rightarrow \{C'\}\,\langle I\rangle\} \times \mathbb{E} \longrightarrow \mathbb{B}$ and call them *verification condition providers* (vc-providers, for short). The truth value $\mathfrak{C}(C, f)$ indicates whether loop C with invariant I satisfies a verification condition w.r.t. postexpectation f. Now fix a vc-provider $\mathfrak{C}$ and some $\mathcal{T} \in \{\mathrm{dwp}, \mathrm{awp}\}$ for the remainder of this section.

Definition 5.1. Let $C \in \mathsf{pGCL}$ and $f \in \mathbb{E}$. The *verification condition* $\mathrm{vc}_{\mathfrak{C}}^{\mathcal{T}}[\![C]\!](f) \in \{\mathsf{true}, \mathsf{false}\}$ of C w.r.t. f (and $\mathfrak{C}$ and $\mathcal{T}$) is defined by induction on C in Figure 11. △

Intuitively, $\mathrm{vc}_{\mathfrak{C}}^{\mathcal{T}}[\![C]\!](f)$ is true if and only if all loops occurring in C satisfy the verification condition given by the vc-provider $\mathfrak{C}$. Notice that, for the sequential composition $C_1 \fatsemi C_2$, we employ the auxiliary transformer $\mathcal{T}^*$ on C_2 to determine the postexpectation for the vc of C_1 since we reason with the annotated loop invariants instead of least fixpoints. The verification condition for the body C' of a loop $\mathtt{while}\,\varphi \rightarrow \{C'\}\,\langle I\rangle$ is taken w.r.t. postexpectation I since, for nested loops, an inner loop's verification condition depends on the outer loop's invariant (see Example 5.3). We are interested in vc-providers $\mathfrak{C}$ so that validity of $\mathrm{vc}_{\mathfrak{C}}^{\mathcal{T}}[\![C]\!](f)$ does indeed imply that $\mathcal{T}^*[\![C]\!](f)$ bounds $\mathcal{T}[\![C]\!](f)$:

Definition 5.2. We say that $\mathfrak{C}$ *yields upper bounds* for $\mathcal{T} \in \{\mathrm{dwp}, \mathrm{awp}\}$, if

$$\text{for all } C \in \mathsf{pGCL} \text{ and all } f \in \mathbb{E}\colon \quad \mathrm{vc}_{\mathfrak{C}}^{\mathcal{T}}[\![C]\!](f) \text{ implies } \mathcal{T}[\![C]\!](f) \sqsubseteq \mathcal{T}^*[\![C]\!](f)\,.$$

Analogously, we say that $\mathfrak{C}$ *yields lower bounds* for $\mathcal{T} \in \{\mathrm{dwp}, \mathrm{awp}\}$, if

$$\text{for all } C \in \mathsf{pGCL} \text{ and all } f \in \mathbb{E}\colon \quad \mathrm{vc}_{\mathfrak{C}}^{\mathcal{T}}[\![C]\!](f) \text{ implies } \mathcal{T}[\![C]\!](f) \sqsupseteq \mathcal{T}^*[\![C]\!](f)\,. \qquad \triangle$$

It is useful to note that, since both $\mathrm{dwp}[\![C]\!]$ and $\mathrm{awp}[\![C]\!]$ are monotonic (see [Batz et al. 2023a, Appendix B]), it actually suffices to show that $\mathfrak{C}$ yields upper (resp. lower) bounds for loops in order to conclude that $\mathfrak{C}$ yields upper (resp. lower) bounds for *all* pGCL programs. See [Batz et al. 2023a, Appendix B] for details.

Example 5.3 (An unbounded nested loop). Recall from Theorem 4.7 that dwp-superinvariants establish upper bounds on demonic weakest preexpectations of loops. Using [Batz et al. 2023a, Lemma B.2], it is thus straightforward to show that the vc-provider SUPERINV given by

$$\mathsf{SUPERINV}(\mathtt{while}\,\varphi \rightarrow \{C'\}\,\langle I\rangle, f) \;=\; \mathsf{true} \qquad \text{iff} \qquad [\varphi]\cdot \mathrm{dwp}^*[\![C']\!](I) + [\neg\varphi]\cdot f \;\sqsubseteq\; I$$

yields upper bounds for dwp. Notice that, for the loop body C', we consider the auxiliary transformer dwp^* to enable reasoning about nested loops. This comes at the cost of the typical dependency between inner- and outer invariants: $\mathrm{dwp}^*[\![C']\!](I)$ must not approximate $\mathrm{dwp}[\![C']\!](I)$ too coarsely, i.e., if C' itself contains loops, then the inner loop's superinvariants must be tight enough for the outer I to satisfy the above superinvariant condition. Now consider program C from Figure 12, which we annotate for determining $\mathrm{dwp}^*[\![C]\!](x)$. Program C contains a nested loop. On every iteration, the outer loop, which we denote by C_1, nondeterministically chooses a value for *inc* and a probability for p. It then flips a fair coin and either terminates or executes the inner loop, which we denote by C_2. Loop C_2 depends on the choices made by C_1: It keeps flipping a coin with bias p and either terminates or keeps incrementing x by *inc*.

The annotations yield the conditions for $\mathrm{vc}_{\mathsf{SUPERINV}}^{\mathrm{dwp}}[\![C]\!](x)$ to be true. Recall that both loops must satisfy their respective verification conditions: The postexpectation for the outer loop C_1 is x, which

```
$^{\mathrm{dwp}^*}/\!\!/\!\!/ \; 3y \sqcap z$
   $c := 1 \fatsemi x := 0$
$^{\mathrm{dwp}^*}/\!\!/\!\!/ \; x + [c \neq 0] \cdot (3y \sqcap z)$                      (the dwp$^*$ of a loop is given by its invariant)
   while $c \neq 0 \rightarrow$ {
      $^{\mathrm{dwp}^*}/\!\!/\!\!/ \; h[p, inc \mapsto 1/4, y] \sqcap h[p, inc \mapsto 1/2, z]$          (this is dwp$^*$⟦loop body⟧$(I)$)
         if true → {  $^{\mathrm{dwp}^*}/\!\!/\!\!/ \; h[p, inc \mapsto 1/4, y]$   $inc := y \fatsemi p := 1/4$   $/\!\!/\!\!/ \; h$  }
         □ true → {  $^{\mathrm{dwp}^*}/\!\!/\!\!/ \; h[p, inc \mapsto 1/2, z]$   $inc := z \fatsemi p := 1/2$   $/\!\!/\!\!/ \; h$  } ⨟
      $^{\mathrm{dwp}^*}/\!\!/\!\!/ \; 1/2 \cdot I[c \mapsto 0] + 1/2 \cdot I'[c' \mapsto 1]$                    (denote this expectation by $h$)
         { $^{\mathrm{dwp}^*}/\!\!/\!\!/ \; I[c \mapsto 0]$   $c := 0$   $/\!\!/\!\!/ \; I$}
         [1/2]
         { $^{\mathrm{dwp}^*}/\!\!/\!\!/ \; I'[c' \mapsto 1]$   $c' := 1 \fatsemi$ $^{\mathrm{dwp}^*}/\!\!/\!\!/ \; I'$  while $c' \neq 0 \rightarrow$ {{ $c' := 0$ } [$p$] { $x := x + inc$ }} ⟨$I'$⟩   $/\!\!/\!\!/ \; I$}
         $/\!\!/\!\!/ \; I$                                   (postexpectation for a loop's body is the loop's invariant)
   } ⟨$x + [c \neq 0] \cdot (3y \sqcap z)$⟩                                    (denote this invariant by $I$)
   $/\!\!/\!\!/ \; x$                                                          ($x$ is the postexpectation)
```

Fig. 12. Program C with annotations for dwp*⟦C⟧(x). Invariant I' of the inner loop is $I' = [p > 0] \cdot \left([c' \neq 0] \cdot \left(\frac{p \cdot x - p \cdot inc + inc}{p} \right) + [c' = 0] \cdot x + [c \neq 0] \cdot (3y \sqcap z) \right)$. Annotations for the inner loop are omitted.

yields the condition $\mathsf{SUPERINV}(C_1, x)$. Moreover, the postexpectation for the inner loop C_2 is I — the invariant of the *outer* loop —, which yields the condition $\mathsf{SUPERINV}(C_2, I)$. Both conditions can be shown to be true. Since SUPERINV yields upper bounds for dwp, we thus get

$$\mathrm{dwp}[\![C]\!](x) \;\sqsubseteq\; \mathrm{dwp}^*[\![C]\!](x) \;=\; 3y \sqcap z\,,$$

i.e., the minimal expected final value of x is upper-bounded by the minimum of $3y$ and z. We will see in the next section how the validity of $\mathrm{vc}^{\mathrm{dwp}}_{\mathsf{SUPERINV}}[\![C]\!](x)$ and the annotation of C_1's body yield a correct-by-construction determinization of C which preserves this bound. △

6 DETERMINIZATIONS OF pGCL PROGRAMS FROM WEAKEST PREEXPECTATIONS

In this section, we formalize our approach for obtaining determinizations of pGCL programs from weakest preexpectations based on our parametric verification conditions from Section 5.

First, recall from Section 4 that our approach is based on (pointwise) comparisons of intermediate preexpectations obtained from applying the rules in Figure 7. To formalize this, we define two functions $\preceq, \succeq \colon \mathbb{E} \times \mathbb{E} \longrightarrow \mathbb{P}$ given by

$$\preceq(f, g) = \lambda\sigma. \begin{cases} \text{true}, & \text{if } f(\sigma) \leq g(\sigma) \\ \text{false}, & \text{otherwise} \end{cases} \quad \text{and} \quad \succeq(f, g) = \lambda\sigma. \begin{cases} \text{true}, & \text{if } f(\sigma) \geq g(\sigma) \\ \text{false}, & \text{otherwise}\,. \end{cases}$$

We often write $f \preceq g$ and $f \succeq g$ instead of $\preceq(f, g)$ and $\succeq(f, g)$. In words, $f \preceq g$ (resp. $f \succeq g$) yields a predicate which evaluates to true on state σ iff $f(\sigma)$ is upper- (resp. lower-) bounded by $g(\sigma)$. These predicates form our sought-after strengthenings of guards to resolve nondeterminism.

Based on the above notion, we define the following expectation-based program transformation:

Definition 6.1. Let $\bowtie \in \{\preceq, \succeq\}$ and $\mathcal{T} \in \{\mathrm{dwp}, \mathrm{awp}\}$. The program transformer

$$\mathrm{trans}^{\mathcal{T}}_{\bowtie} \colon \mathrm{pGCL} \times \mathbb{E} \rightarrow \mathrm{pGCL}$$

is defined by induction on pGCL in Figure 13. △

C	$\mathsf{trans}^{\mathcal{T}}_{\bowtie}(C, f)$
`skip`	`skip`
$x := E$	$x := E$
$C_1 \fatsemi C_2$	$\mathsf{trans}^{\mathcal{T}}_{\bowtie}(C_1, \mathcal{T}^*[\![C_2]\!](f)) \fatsemi \mathsf{trans}^{\mathcal{T}}_{\bowtie}(C_2, f)$
$\mathtt{if}\, \varphi_1 \to \{C_1\}$ $\square\; \varphi_2 \to \{C_2\}$	$\mathtt{if}\, \varphi_1 \wedge (\varphi_2 \Rightarrow \mathcal{T}^*[\![C_1]\!](f) \bowtie \mathcal{T}^*[\![C_2]\!](f)) \to \{\mathsf{trans}^{\mathcal{T}}_{\bowtie}(C_1, f)\}$ $\square\; \varphi_2 \wedge (\varphi_1 \Rightarrow \mathcal{T}^*[\![C_2]\!](f) \bowtie \mathcal{T}^*[\![C_1]\!](f)) \to \{\mathsf{trans}^{\mathcal{T}}_{\bowtie}(C_2, f)\}$
$\{\, C_1 \,\}\; [p]\; \{\, C_2 \,\}$	$\{\, \mathsf{trans}^{\mathcal{T}}_{\bowtie}(C_1, f) \,\}\; [p]\; \{\, \mathsf{trans}^{\mathcal{T}}_{\bowtie}(C_2, f) \,\}$
$\mathtt{while}\, \varphi \to \{C'\}\, \langle I \rangle$	$\mathtt{while}\, \varphi \to \{\mathsf{trans}^{\mathcal{T}}_{\bowtie}(C', I)\}\, \langle I \rangle$

Fig. 13. Inductive definition of the program transformer trans.

We often abbreviate $\mathsf{trans}^{\mathsf{dwp}}_{\preceq}(C, f)$ and $\mathsf{trans}^{\mathsf{awp}}_{\succeq}(C, f)$ by $\mathsf{dtrans}(C, f)$ (called the *demonic transformation* of C w.r.t. f) and $\mathsf{atrans}(C, f)$ (called the *angelic transformation* of C w.r.t. f), respectively. Let us go over the rules in Figure 13. `skip` statements and assignments are never transformed. For the sequential composition $C_1 \fatsemi C_2$, we transform C_2 w.r.t. f and C_1 w.r.t. to $\mathcal{T}^*[\![C_2]\!](f)$. Notice that we employ the auxiliary transformer from Section 5 since our determinizations are guided by the annotated loop invariants. The guarded choice is the most interesting case. Recall from our informal description from Section 4 that we construct strengthenings for the guards φ_1 and φ_2 by comparing the intermediate preexpectations $\mathcal{T}^*[\![C_1]\!](f)$ and $\mathcal{T}^*[\![C_2]\!](f)$ of the two branches. Whenever *both* φ_1 and φ_2 hold, there is nondeterminism that is to be resolved, which is realized by applying function $\bowtie\, \in \{\preceq, \succeq\}$ to the intermediate preexpectations. Here, $\Rightarrow$ is the standard implication between predicates, i.e, $\varphi \Rightarrow \psi$ is false on state σ iff $\sigma \models \varphi$ and $\sigma \not\models \psi$. We refer to Section 4.2 for an illustrative example, where the corresponding intermediate preexpectations are highlighted. For the probabilistic choice, we simply transform the two respective branches. Finally, for loops, we transform the loop body w.r.t. the annotated loop invariant (cf. Example 6.6).

The above transformations yield implementations of the program that is to be transformed:

Theorem 6.2. *Let* $\bowtie\, \in \{\preceq, \succeq\}$, $\mathcal{T} \in \{\mathsf{dwp}, \mathsf{awp}\}$, $C \in \mathsf{pGCL}$, $f \in \mathbb{E}$. *We have* $\mathsf{trans}^{\mathcal{T}}_{\bowtie}(C, f) \multimap C$.

Proof. By induction on C. See [Batz et al. 2023a, Appendix A] for details. □

Hence, $\mathsf{dwp}[\![C]\!](f) \sqsubseteq \mathsf{dwp}[\![\mathsf{dtrans}(C, f)]\!](f)$ and $\mathsf{awp}[\![C]\!](f) \sqsupseteq \mathsf{awp}[\![\mathsf{atrans}(C, f)]\!](f)$, hold by Lemma 4.3, i.e., naturally, resolving nondeterminism by applying the program transformations increases (decreases) minimal (maximal) expected outcomes of C.

As demonstrated in Section 4.2, the program transformations generally yield programs which are still nondeterministic. This remaining nondeterminism can, however, be resolved in an arbitrary manner in the sense that $\mathsf{dtrans}(C, f)$ and $\mathsf{atrans}(C, f)$ yield *optimal permissive determinizations* (cf. Section 4.2) w.r.t. the auxiliary transformers dwp^* and awp^*:

Theorem 6.3. *Let* $C, C' \in \mathsf{pGCL}$ *and* $f \in \mathbb{E}$. *We have*

$$C' \multimap \mathsf{dtrans}(C, f) \qquad \textit{implies} \qquad \mathsf{dwp}^*[\![C']\!](f) \;=\; \mathsf{dwp}^*[\![C]\!](f)\,.$$

Moreover, we have

$$C' \multimap \mathsf{atrans}(C, f) \qquad \textit{implies} \qquad \mathsf{awp}^*[\![C']\!](f) \;=\; \mathsf{awp}^*[\![C]\!](f)\,.$$

Proof. By induction on C. See [Batz et al. 2023a, Appendix A] for details. □

In particular, if C is loop-free then $\mathrm{dwp}^*[\![C]\!](f)$ and $\mathrm{dwp}[\![C]\!](f)$ (resp. $\mathrm{awp}^*[\![C]\!](f)$ and $\mathrm{awp}[\![C]\!](f)$) coincide, so the above theorem indeed yields that our transformation is *optimal and correct for loop-free programs*. In the presence of loops, we need to ensure that the respective annotated invariants soundly bound the sought-after preexpectations. For that, recall from Section 5 that we introduced vc-providers $\mathfrak{C}$ and from Section 4.3 that our idea is to construct determinizations which preserve validity of the so-defined verification conditions. This motivates the following definition:

Definition 6.4. We say that dtrans *preserves* $\mathfrak{C}$ if for all $C, C' \in \mathrm{pGCL}$ and all $f \in \mathbb{E}$,

$$\mathrm{vc}^{\mathrm{dwp}}_{\mathfrak{C}}[\![C]\!](f) \text{ and } C' \multimap \mathrm{dtrans}(C, f) \quad \text{implies} \quad \mathrm{vc}^{\mathrm{dwp}}_{\mathfrak{C}}[\![C']\!](f)\,.$$

Moreover, we say that atrans *preserves* $\mathfrak{C}$ if for all $C, C' \in \mathrm{pGCL}$ and all $f \in \mathbb{E}$,

$$\mathrm{vc}^{\mathrm{awp}}_{\mathfrak{C}}[\![C]\!](f) \text{ and } C' \multimap \mathrm{atrans}(C, f) \quad \text{implies} \quad \mathrm{vc}^{\mathrm{awp}}_{\mathfrak{C}}[\![C']\!](f)\,. \qquad \triangle$$

In words, all implementations C' of $\mathrm{dtrans}(C, f)$ (including all determinizations and $\mathrm{dtrans}(C, f)$ itself) must preserve validity of the verification condition given by the vc-provider $\mathfrak{C}$ (analogously for $\mathrm{atrans}(C, f)$). Hence, if $\mathfrak{C}$ yields upper bounds for dwp (resp. lower bounds for awp), then the respective program transformations *preserve* the bounds obtained from the annotated loop invariants, i.e., $\mathrm{dwp}^*[\![C]\!](f)$ and $\mathrm{awp}^*[\![C]\!](f)$, respectively, which is the main result of this section:

Theorem 6.5. *Assume that* dtrans preserves $\mathfrak{C}$. *If* $\mathfrak{C}$ *yields upper bounds for* dwp, *then for all* $C, C' \in \mathrm{pGCL}$ *and all* $f \in \mathbb{E}$,

$$\mathrm{vc}^{\mathrm{dwp}}_{\mathfrak{C}}[\![C]\!](f) \textit{ and } C' \multimap \mathrm{dtrans}(C, f) \qquad \textit{implies} \qquad \mathrm{dwp}[\![C]\!](f) \sqsubseteq \mathrm{dwp}[\![C']\!](f) \sqsubseteq \mathrm{dwp}^*[\![C]\!](f)\,.$$

Dually, assume that atrans preserves $\mathfrak{C}$. *If* $\mathfrak{C}$ *yields lower bounds for* awp, *then for all* $C, C' \in \mathrm{pGCL}$ *and all* $f \in \mathbb{E}$,

$$\mathrm{vc}^{\mathrm{awp}}_{\mathfrak{C}}[\![C]\!](f) \textit{ and } C' \multimap \mathrm{atrans}(C, f) \qquad \textit{implies} \qquad \mathrm{awp}[\![C]\!](f) \sqsupseteq \mathrm{awp}[\![C']\!](f) \sqsupseteq \mathrm{awp}^*[\![C]\!](f)\,.$$

Proof. We prove the claim for dwp. The reasoning for awp is analogous. First, $\mathrm{dwp}[\![C]\!](f) \sqsubseteq \mathrm{dwp}[\![C']\!](f)$ holds by Theorem 6.2 and Lemma 4.3. Moreover, since dtrans preserves $\mathfrak{C}$, we have $\mathrm{vc}^{\mathrm{dwp}}_{\mathfrak{C}}[\![C']\!](f)$. Hence, since $\mathfrak{C}$ yields upper bounds for dwp, we get

$$\mathrm{dwp}[\![C']\!](f) \overset{\text{Def. 5.2}}{\sqsubseteq} \mathrm{dwp}^*[\![C']\!](f) \overset{\text{Thm. 6.3}}{=} \mathrm{dwp}^*[\![C]\!](f)\,. \qquad \square$$

Example 6.6. Recall the vc-provider SUPERINV from Example 5.3, which yields upper bounds for dwp. Using Theorem 6.3, it is immediate that dtrans preserves SUPERINV. Now reconsider program C from Figure 12. Since dwp^* is obtained using the annotated loop invariants, the highlighted intermediate preexpectations are obtained in a purely syntactic manner. Applying minor arithmetic simplifications to the guard strengthenings obtained from these highlighted preexpectations, we thus also obtain $\mathrm{dtrans}(C, x)$ in a purely syntactic manner:

```
$c := 1 ⨾ x := 0$
while $c \neq 0 \to \{$
    if $6y \leq z \to \{inc := y ⨾ p := 1/4\}$
     $\square$ $6y \geq z \to \{inc := z ⨾ p := 1/2\}$ ⨾
    $\{c := 0\}$
    $[1/2]$
    $\{c' := 1 ⨾$ while $c' \neq 0 \to \{\{c' := 0\}\ [p]\ \{x := x + inc\}\}\ \langle I' \rangle\ \}$
$\}\ \langle x + [c \neq 0] \cdot (3y \sqcap z) \rangle$
```

The above nested loop still behaves nondeterministically whenever $6y = z$ holds at the beginning of an outer loop iteration. However, thanks to Theorem 6.5, *any* determinization C' of $\mathsf{dtrans}(C, x)$ (obtained by, e.g., turning the first inequality $6y \leq z$ into a strict one) satisfies

$$\mathsf{dwp}[\![C]\!](x) \;\sqsubseteq\; \mathsf{dwp}[\![C']\!](x) \;=\; \mathsf{dwp}^*[\![C]\!](x) \;=\; 3y \sqcap z,$$

i.e, the expected final value of x of *any* determinization C' is upper-bounded by $3y \sqcap z$. △

6.1 Completeness

Recall from Section 4.4 that our approach is inherently incomplete when it comes to finding determinizations which guarantee lower bounds on expected outcomes of pGCL programs. Yet, Theorem 6.5 yields a sufficient condition for our approach to be complete for *subclasses* of problem instances. Let $\Pi \subseteq \mathsf{pGCL}$ and $\Xi \subseteq \mathbb{E}$. We call a vc-provider $\mathfrak{C}$ *demonically (resp. angelically) complete* w.r.t. (Π, Ξ) if, (i) vc yields upper (resp. lower) bounds for dwp (resp. awp) and (ii) for all $C \in \Pi$ and $f \in \Xi$, program C can be annotated with invariants in such a way that $\mathsf{vc}_{\mathfrak{C}}^{\mathsf{dwp}}[\![C]\!](f)$ (resp. $\mathsf{vc}_{\mathfrak{C}}^{\mathsf{awp}}[\![C]\!](f)$) holds and $\mathsf{dwp}[\![C]\!](f) = \mathsf{dwp}^*[\![C]\!](f)$ (resp. $\mathsf{awp}[\![C]\!](f) = \mathsf{awp}^*[\![C]\!](f)$).

Theorem 6.7. *Let $\mathfrak{C}$ be demonically complete w.r.t.* (Π, Ξ) *and preserved by* dtrans. *Then, for all $C \in \Pi$ and $f \in \Xi$, there exist invariant annotations for C such that for all $C' \in \mathsf{pGCL}$,*

$$C' \multimap \mathsf{dtrans}(C, f) \qquad \textit{implies} \qquad \mathsf{dwp}[\![C]\!](f) \;=\; \mathsf{dwp}[\![C']\!](f)\;.$$

Analogously, let $\mathfrak{C}$ be angelically complete w.r.t. (Π, Ξ) *and preserved by* atrans. *Then, for every $C \in \Pi$ and $f \in \Xi$, there exist invariant annotations for C such that for all $C' \in \mathsf{pGCL}$,*

$$C' \multimap \mathsf{atrans}(C, f) \qquad \textit{implies} \qquad \mathsf{awp}[\![C]\!](f) \;=\; \mathsf{awp}[\![C']\!](f)\;.$$

As mentioned in Section 4.4, Theorem 6.7 immediately yields that our approach is complete w.r.t. *upper* bounds: the vc-provider SUPERINV from Examples 5.3 and 6.6 is demonically complete w.r.t. $(\mathsf{pGCL}, \mathbb{E})$ by annotating each loop occurring in some program C with its respective least fixpoint (cf. Figure 7). We provide a complete subclass for *lower* bounds in the next section.

6.2 Instances of our Framework

6.2.1 Upper-Bounded Determinizations. We restate the vc-provider SUPERINV for later reference.

Theorem 6.8. *The* vc*-provider* SUPERINV *yields upper bounds for* dwp *and is preserved by* dtrans*:*

$$\mathsf{SUPERINV}(\mathtt{while}\,\varphi \to \{C_{\text{body}}\}\,\langle I\rangle\,, f) \;=\; \mathsf{true} \qquad \text{iff} \qquad [\varphi]\cdot \mathsf{dwp}^*[\![C_{\text{body}}]\!](I) + [\neg\varphi]\cdot f \;\sqsubseteq\; I$$

6.2.2 Lower-Bounded Determinizations. Lower-bounding weakest preexpectations of loops is more involved. There exist specialized proof rules, which typically restrict the expressive power of the invariants, the postexpectation, and the termination behavior of the loop under consideration.

We call program C *demonically almost-surely terminating* (dAST, for short) if $\mathsf{dwp}[\![C]\!](1) = 1$. Moreover, we call C *positively demonically almost-surely terminating* (dPAST, for short) if C's expected runtime (in the sense of [Kaminski et al. 2016]) is finite for all initial states. Finally, we define a loop $\mathtt{while}\,\varphi \to \{C_{\text{body}}\}\,\langle I\rangle$ to be *suitable for optional stopping w.r.t. f* in [Batz et al. 2023a, Appendix B].

With these notions at hand, we obtain instances for lower-bounded determinizations:

Theorem 6.9. *The following* vc*-providers yield lower bounds for* awp *and are preserved by* atrans*:*

(1) dASTSUBINV *which for $C = \mathtt{while}\,\varphi \to \{C_{\text{body}}\}\,\langle I\rangle$ and $f \in \mathbb{E}$ is given by*

$$\mathsf{dASTSUBINV}(C, f) = \mathsf{true} \qquad \textit{iff} \qquad \begin{array}{c}[\varphi]\cdot \mathsf{awp}^*[\![C_{\text{body}}]\!](I) + [\neg\varphi]\cdot f \sqsupseteq I, C \textit{ is dAST,}\\ I \textit{ and } f \textit{ are bounded by some constant } b\;.\end{array}$$

(2) dPASTSUBINV *which for* $C = \mathtt{while}\,\varphi \to \{C_{\text{body}}\}\,\langle I\rangle$ *and* $f \in \mathbb{E}$ *is given by*

$$\mathsf{dPASTSUBINV}(C, f) = \mathsf{true} \qquad \textit{iff} \qquad \begin{array}{c}[\varphi]\cdot \mathsf{awp}^*[\![C_{\text{body}}]\!](I) + [\neg\varphi]\cdot f \sqsupseteq I,\ C \textit{ is dPAST,} \\ C \textit{ is suitable for optional stopping w.r.t. } f\ .\end{array}$$

Proof. Preservation by atrans follows immediately from Lemma 4.3 and Theorem 6.3 and the fact that implementations C' of C remain dAST and dPAST, respectively. The fact that the providers yield lower bounds for awp follows from [Hark et al. 2020, Theorem 38] (see Section 6.2.3). □

The vc-provider dASTSUBINV is angelically complete w.r.t.

$$\big(\{C \in \mathsf{pGCL} \mid C \text{ is dAST}\}, \{f \in \mathbb{E} \mid f \text{ bounded by some } b \in \mathbb{R}_{\geq 0}\}\big)$$

by annotating every loop in a given program C by its respective least fixpoint, which is also bounded by [Batz et al. 2023a, Theorem B.1.2]. Theorem 6.7 thus yields completeness of our approach for this class of problem instances. Determining the class for which dPASTSUBINV is complete is an open problem. Notice that dPASTSUBINV does not require I or f to be bounded which comes at the cost of requiring the stronger termination criterion dPAST and being suitable for optional stopping.

6.2.3 *Lower Bounds for* awp.

The fact that dPASTSUBINV yields lower bounds for awp (cf. Theorem 6.9.2), follows from a result from the literature which applies to *deterministic* pGCL programs [Hark et al. 2020, Theorem 38]. Hark et al. [2020] leave an extension of this rule for *non*deterministic programs for future work. Our results from Section 6 yield such an extension, i.e., a novel proof rule for lower bounds on awp's if possibly nondeterministic loops. First, consider non-nested loops:

Theorem 6.10. *Let* $C = \mathtt{while}\,\varphi \to \{C_{\text{body}}\}\,\langle I\rangle$ *and* $f \in \mathbb{E}$ *with* C_{body} *loop-free. If*

(1) $[\varphi]\cdot \mathsf{awp}[\![C_{\text{body}}]\!](I) + [\neg\varphi]\cdot f \sqsupseteq I$, *and*
(2) C *is dPAST, and*
(3) C *is suitable for optional for stopping w.r.t.* f *(cf. [Batz et al. 2023a, Appendix B]),*

then I *lower-bounds* $\mathsf{awp}[\![C]\!](f)$*, i.e.,*

$$\mathsf{awp}[\![C]\!](f) \ \sqsupseteq\ I\ .$$

Proof. First notice that the conjunction of Conditions 1, 2, and 3 from above is equivalent to dPASTSUBINV(C, f) = true (cf. Theorem 6.9.2). Since atrans preserves dPASTSUBINV, there is a determinization C' of C satisfying dPASTSUBINV(C', f). Hence [Hark et al. 2020, Theorem 38] *does* apply to this deterministic program C' and we get

$$\mathsf{awp}[\![C]\!](f) \ \overset{\text{Lem. 4.3}}{\sqsupseteq}\ \mathsf{wp}[\![C']\!](f) \ \overset{\text{[Hark et al. 2020, Theorem 38]}}{\sqsupseteq}\ I\ .$$

□

Example 6.11. Let $f = x$ and $I = [c = 1]\cdot(x+2) + [c \neq 1]\cdot x$ and let C be the loop

$$\mathtt{while}\, c = 1 \to \{\{\, c := 0\,\}\ [0.5]\ \{\,\mathtt{if}\,\mathsf{true} \to \{x := x+1\} \,\square\, \mathsf{true} \to \{x := x+2\}\,\}\}\,\langle I\rangle\ .$$

We employ Theorem 6.10 to prove that $\mathsf{awp}[\![C]\!](x) \sqsupseteq I$. Regarding Conditions 3 and 2, it is easy to verify that C is dPAST (using, e.g., the ert-calculus [Kaminski et al. 2016]) and suitable for optional stopping w.r.t. to x. For Condition 1, we calculate

$$\begin{aligned} & [c=1]\cdot \mathsf{awp}[\![C_{\text{body}}]\!](I) + [c \neq 1]\cdot x \\ =\ & [c=1]\cdot 0.5\cdot \big(I[c \mapsto 0] + (I[x \mapsto x+1] \sqcup I[x \mapsto x+2])\big) + [c \neq 1]\cdot x \\ =\ & [c=1]\cdot 0.5\cdot \big(x + ((x+3) \sqcup (x+4))\big) + [c \neq 1]\cdot x \\ =\ & [c=1]\cdot 0.5\cdot (2\cdot x + 4) + [c \neq 1]\cdot x \ =\ I \ \sqsupseteq\ I\ . \end{aligned}$$

△

By recursively applying Theorem 6.10 and determinizing inner loops, this generalizes to nested loops:

THEOREM 6.12. *Let $C = \texttt{while}\,\varphi \rightarrow \{C_{\text{body}}\}\,\langle I \rangle$ be a (possibly nested) loop and let $f \in \mathbb{E}$. We have*

$$\mathsf{vc}^{\mathsf{awp}}_{\mathsf{dPASTSUBINV}}[\![C]\!](f) \qquad \textit{implies} \qquad \mathsf{awp}[\![C]\!](f) \sqsupseteq I\,.$$

7 CASE STUDIES

In this section, we apply our framework to synthesize bound-guaranteeing determinizations of pGCL programs, all of which can be understood as countably infinite-state MDPs.

7.1 Game of Nim

We consider a variant of the game Nim (e.g. [Wikipedia 2023]), a 2-player zero-sum game which goes as follows: N tokens are placed on a table. The players take turns; in each turn, the player has to remove 1, 2, or 3 tokens from the table. The first player to remove the last token looses the game.

Suppose we are interested in finding a strategy performing reasonably well against an opponent that plays randomly. We model this situation as the pGCL program C_{Nim} in Figure 2 on page 3. Variable x counts the number of tokens that have been removed so far. Variable $turn \in \{1,2\}$ indicates which player takes the next token(s). The randomized opponent is player 1. If the program terminates in state where $turn = i$, then player i wins the game.

We claim that the controllable player 2 can guarantee the following when starting the game with $x < N$ removed tokens:

- If it's player 1's turn and $x + 1 \equiv_4 N$ (i.e., $x + 1 - N$ is a multiple of 4), or if it's player 2's turn and $x + 1 \not\equiv_4 N$, then player 2 *can win with probability one.*
- If it' player 1's turn but $x + 1 \not\equiv_4 N$, then player 2 can still *win with probability* $\geq 2/3$.

Formally, the *maximal* probability that player 2 wins is given by $\mathsf{awp}[\![C_{Nim}]\!]([turn = 2])$. We wish to find a determinization C''_{Nim} of C_{Nim} — a strategy for player 2 — which guarantees the above lower bound on player 2's probability to win, i.e.,

$$\begin{aligned}&[x < N] \cdot \Big([turn = 1] \cdot \big([x+1 \equiv_4 N] + \tfrac{2}{3} \cdot [x+1 \not\equiv_4 N] \big) \,+\, [turn = 2] \cdot [x+1 \not\equiv_4 N] \Big) \\ \sqsubseteq\;& \mathsf{wp}[\![C''_{Nim}]\!]([turn = 2])\,. \qquad (\dagger)\end{aligned}$$

For that, we annotate C_{Nim} with the subinvariant I_{Nim}:

$$\begin{aligned}I_{Nim} \;\coloneqq\;& [x < N] \cdot \Big([turn = 1] \cdot \big([x+1 \equiv_4 N] + \tfrac{2}{3} \cdot [x+1 \not\equiv_4 N] \big) \,+\, [turn = 2] \cdot [x+1 \not\equiv_4 N] \Big) \\ &+\, [x \geq N] \cdot [turn = 2] \qquad \text{(see [Batz et al. 2023a] for a proof of subinvariance)}\end{aligned}$$

Applying $\mathsf{atrans}(C_{Nim}, [turn = 2])$, we obtain $C'_{Nim} \multimap C_{Nim}$ shown in Figure 3 on page 3. Since C_{Nim} is clearly dAST (it iterates at most $N - x$ times), we may employ the vc-provider from Theorem 6.9.1 to conclude that C'_{Nim} represents a permissive controller for player 2 which guarantees the above bound, i.e., *each* deterministic $C''_{Nim} \multimap C'_{Nim}$ satisfies the requirement (†).

7.2 Optimal Gambling

7.2.1 Maximizing the Winning Probability. Consider the following gambling situation. A gambler has to collect N tokens to win a prize. The game is played in rounds: In each round, the gambler has to choose between flipping two coins with different biases. Suppose the coins yield heads with probability p and q, respectively. If the result of the coin flip is tails, then the game is immediately lost. Otherwise, the gambler wins one token in case of the bias-p coin, and two tokens in case of the bias-q coin. The goal is to maximize the probability of winning given an initial budget c of tokens,

```
while $c < N \wedge a = 0 \rightarrow \{$
    if true
        $\rightarrow \{ c := c + 1 \}\ [p]\ \{ a := 1 \}$
    if true
        $\rightarrow \{ c := c + 2 \}\ [q]\ \{ a := 1 \}$
$\}\ \langle I_{Gamb} \rangle$
```

Fig. 14. Program C_{Gamb} modeling a gamble.

```
while $c < N \wedge a = 0 \rightarrow \{$
    if $(q \leq p^2) \vee (p^2 < q < p \wedge N - c \text{ is odd})$
        $\rightarrow \{ c := c + 1 \}\ [p]\ \{ a := 1 \}$
    if $(p \leq q) \vee (q = p^2) \vee (p^2 < q < p \wedge N - c \geq 2)$
        $\rightarrow \{ c := c + 2 \}\ [q]\ \{ a := 1 \}$
$\}\ \langle I_{Gamb} \rangle$
```

Fig. 15. Program C'_{Gamb} encoding all optimal strategies.

and the game parameters p, q, and N. We briefly describe the optimal way to play this game (note that the probability to win two tokens in consecutive rounds with the bias-p coin is p^2):

- If $p^2 \geq q$, then playing with the bias-p coin is optimal.
- Similarly, if $p \leq q$, then the gambler should always choose the bias-q coin.
- Otherwise $p^2 < q < p$. In this case, the optimal choice depends on the current budget c: If only $N - c = 1$ token is needed to win the game, then it is better to choose the bias-p coin. More generally, if $N - c$ is an odd number, then the best strategy is to play once with the bias-p coin and up to $(N-c-1)/2$ times with the bias-q coin (the order is irrelevant).

The gamble is readily modeled as the pGCL program C_{Gamb} in Figure 14 (we assume that $c, N \in \mathbb{N}$, $a \in \{0, 1\}$, and $p, q \in [0, 1]$). Note that the game is lost as soon as $a = 1$. Finding an *optimal* gambling strategy that *minimizes* the probability of losing thus amounts to finding

$$\text{a deterministic } C^{det}_{Gamb} \multimap C_{Gamb} \qquad \text{such that} \qquad \text{wp}[\![C^{det}_{Gamb}]\!]([a = 1]) \;=\; \text{dwp}[\![C_{Gamb}]\!]([a = 1]) \;.$$

For that, we annotate C_{Gamb} with the superinvariant I_{Gamb}:

$$I_{Gamb} \;\coloneqq\; [a = 0] \cdot \Bigg(1 \;-\; \Big([p \leq q] \cdot q^{\lceil (N-c)/2 \rceil} \;+\; \big[q \leq p^2\big] \cdot p^{N-c} \\ +\; \big[p^2 < q < p\big] \cdot p^{[N-c>0][N-c \text{ is odd}]} \cdot q^{\lfloor (N-c)/2 \rfloor} \Big)\Bigg) \;+\; [a = 1]$$

See [Batz et al. 2023a] for a proof of superinvariance. Now, applying dtrans$(C_{Gamb}, [a = 1])$, we obtain $C'_{Gamb} \multimap C_{Gamb}$ shown in Figure 15. We may thus apply the vc-provider from Theorem 6.8 to conclude that $\text{dwp}[\![C_{Gamb}]\!]([a = 1]) \sqsubseteq \text{dwp}[\![C'_{Gamb}]\!]([a = 1])$. In fact, it can be shown that I_{Gamb} is the *exact* least fixpoint[7] of C_{Gamb}, so we obtain that

> *C'_{Gamb} is a correct-by-construction optimal implementation of C_{Gamb} w.r.t. minimizing the probability that $a = 1$ holds when the program terminates.*

Observe that program C'_{Gamb} indeed represents the informal description of an optimal strategy given above. C'_{Gamb} is still nondeterministic, e.g. if $p^2 < q < p$ and $N - c = 3$, then both choices are enabled. However, Theorem 6.5 ensures that *every* determinization $C^{det}_{Gamb} \multimap C'_{Gamb}$ satisfies $\text{wp}[\![C^{det}_{Gamb}]\!]([a = 1]) = \text{dwp}[\![C_{Gamb}]\!]([a = 1])$ as desired.

[7]We have $\text{dwp}[\![I_{Gamb}]\!]([a = 1]) \geq I_{Gamb}$ because I_{Gamb} is a dwp-subinvariant of C_{Gamb}, C_{Gamb} is dAST (the loop is executed at most $N - c$ times, regardless of the gambler's choices), and the postexpectation $[a = 1]$ is bounded [Kaminski 2019].

```
$c := 0 ⨟ a := 0 ⨟$
while $a = 0 \to$ {
    if true $\to$ {{ $c := c + 1$ } [$p$] { $a := 1$ }}
    $\square$ true $\to$ {{ $c := c + 2$ } [$q$] { $a := 1$ }}
} $\langle I_{Gamb2} \rangle$
```

Fig. 16. C_{Gamb2}

```
$c := 0 ⨟ a := 0 ⨟$
while $a = 0 \to$ {
    if $2q(1-p) \leq p(1-q)$ $\to$ {{ $c := c + 1$ } [$p$] { $a := 1$ }}
    $\square$ $2q(1-p) \geq p(1-q)$ $\to$ {{ $c := c + 2$ } [$q$] { $a := 1$ }}
} $\langle I_{Gamb2} \rangle$
```

Fig. 17. C_{Gamb2} after transformation, assuming $p, q < 1$.

7.2.2 Maximizing the Expected Payoff. Consider a variant of the above game where the gambler receives their current number of tokens c as a prize once the variable a becomes 0 (modeled by program C_{Gamb2} in Figure 16). We synthesize a strategy to win at least $\frac{p}{1-p} \sqcup \frac{2q}{1-q}$ tokens on expectation (assuming $p, q < 1$). For that, we annotatate the loop with subinvariant

$$I_{Gamb2} \coloneqq [a = 1] \cdot c + [a = 0] \cdot \left(c + \left(\tfrac{p}{1-p} \sqcup \tfrac{2q}{1-q}\right)\right)$$

and show in [Batz et al. 2023a] that we may employ the vc-provider from Theorem 6.9.2. Hence, Theorem 6.5 yields that *any* determinization C''_{Gamb2} of C'_{Gamb2} = atrans(C_{Gamb2}, c) shown in Figure 17 satisfies $\mathrm{wp}[\![C''_{Gamb2}]\!](c) \sqsupseteq \frac{p}{1-p} \sqcup \frac{2q}{1-q}$, i.e., C''_{Gamb2} indeed realizes the lower-bound on the expected payoff of the game we aimed for.

8 RELATED WORK

Strategy Synthesis in Markov Decision Processes. MDPs have a rich mathematical theory [Puterman 1994] and widespread applications across different fields. In machine learning, the well-known *reinforcement learning* problem is typically phrased in terms of MDPs, see e.g. [van Otterlo and Wiering 2012]. However, RL usually does not provide strict guarantees about the resulting strategy. *Exact analysis* of MDPs —i.e., finding *provably* correct strategies as we do in this paper— is one of the primary problems studied in the Probabilistic Model Checking (PMC) community, see [Katoen 2016] and references therein for an overview. PMC tools such as PRISM [Kwiatkowska et al. 2002] or Storm [Hensel et al. 2022] support strategy synthesis for MDPs given in pGCL-like modeling languages. Compact symbolic representations of such strategies have been studied as well [Ashok et al. 2020]. The main difference to our approach is that these tools, and in fact most of the PMC literature, support only *finite* MDPs and work by exploring the full state space. Several subclasses of infinite MDPs have also been studied, including solvency games [Berger et al. 2008], 1-counter and recursive MDPs [Brázdil et al. 2010; Etessami and Yannakakis 2015], or parametric MDPs [Junges et al. 2021]. Some of these works yield efficient algorithms, e.g. deciding if a target state can be reached with probability one in an MDP with one unbounded counter is decidable in PTIME [Brázdil et al. 2010]. Our pGCL programs subsume these models, but their high expressivity comes at the cost of general undecidability.

Beyond PMC, researchers in AI have studied *symbolic dynamic programming* [Boutilier et al. 2001; Sanner et al. 2011], a class of *logic*-based representations and solution methods for MDPs. These methods can be seen as a symbolic variant of value iteration. In contrast to that, our approach is based on programs and uses invariants rather than explicit iteration.

Program Refinement. Our approach is related to program refinement, where, originally, the goal is to refine an abstract model or specification to an executable (non-probabilistic) program [Abrial 2010; Back and von Wright 1998; Lamport 2002]. Later, the concept of program refinement has been generalized to the probabilistic setting [Aouadhi et al. 2019; Hoang et al. 2005; McIver and

Morgan 2005]. Our problem statement can be understood as a program refinement problem: Given a nondeterministic pGCL program C (the abstract model) satisfying some specification (i.e., $\mathrm{dwp}[\![C]\!](f) \sqsubseteq g$ or $\mathrm{awp}[\![C]\!](f) \sqsupseteq g$), refine C to (a determinization) C' which preserves this specification. The aforementioned approaches typically allow for more general specifications, which, however, comes with the loss of mechanizability. Given loop invariants satisfying their respective verification conditions, our approach is highly constructive as we obtain refinements in a syntactic manner. Rather than being a formal system for deriving provably correct algorithms, our approach is thus more tailored to a planning/AI setting, where the nondeterminism models different ways for an agent/adversary to behave, and where one wants to find suitable strategies.

Finally, [Mamouras 2016] uses a variant of Hoare logic to find strategies for (non-stochastic) games. Mamouras [2016], however, works in an uninterpreted and purely qualtitative setting. It is unclear how to adapt this work to the quantitative probabilistic setting.

9 CONCLUSION & FUTURE WORK

We have presented a framework for obtaining strategies for nondeterministic probabilistic programs by means of deductive verification techniques. Several instances of this framework alongside with case studies demonstrate the applicability of our approach. Future work includes mechanizing our techniques in proof assistants [Müller et al. 2017] and deductive verification infrastructures [Schröer et al. 2023]. Moreover, we plan to extend existing invariant synthesis techniques for probabilistic programs [Bao et al. 2022; Batz et al. 2023b; Feng et al. 2017; Katoen et al. 2010] to support nondeterminism. This will yield our approach to produce program-level strategies in a *fully automated manner.* In addition to that, our framework could also be instantiated with more advanced proof rules for loops (see, e.g., [Batz et al. 2021a]), which often yield simpler invariants. Finally, we intend to transfer our technique to non-functional requirements such as expected runtimes [Batz et al. 2023c; Kaminski et al. 2016] or more general *weighted programs* [Batz et al. 2022].

DATA AVAILABILITY STATEMENT

A full version of this article is available online [Batz et al. 2023a].

ACKNOWLEDGMENTS

The authors thank Annabelle McIver and Carroll Morgan for their pointers to existing literature on probabilistic program refinement.

REFERENCES

Jean-Raymond Abrial. 2010. *Modeling in Event-B - System and Software Engineering.* Cambridge University Press. https://doi.org/10.1017/CBO9781139195881

Mohamed Amine Aouadhi, Benoît Delahaye, and Arnaud Lanoix. 2019. Introducing probabilistic reasoning within Event-B. *Softw. Syst. Model.* 18, 3 (2019), 1953–1984. https://doi.org/10.1007/S10270-017-0626-5

Pranav Ashok, Mathias Jackermeier, Pushpak Jagtap, Jan Kretínský, Maximilian Weininger, and Majid Zamani. 2020. dtControl: decision tree learning algorithms for controller representation. In *HSCC.* ACM, 17:1–17:7. https://doi.org/10.1145/3365365.3382220

Ralph-Johan Back and Joakim von Wright. 1998. *Refinement Calculus - A Systematic Introduction.* Springer. https://doi.org/10.1007/978-1-4612-1674-2

Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking.* MIT Press.

Jialu Bao, Nitesh Trivedi, Drashti Pathak, Justin Hsu, and Subhajit Roy. 2022. Data-Driven Invariant Learning for Probabilistic Programs. In *CAV (1) (Lecture Notes in Computer Science, Vol. 13371).* Springer, 33–54. https://doi.org/10.1007/978-3-031-13185-1_3

Kevin Batz, Tom Jannik Biskup, Joost-Pieter Katoen, and Tobias Winkler. 2023a. Programmatic Strategy Synthesis: Resolving Nondeterminism in Probabilistic Programs. https://doi.org/10.48550/ARXIV.2311.06889 arXiv:2311.06889 [cs.LO]

Kevin Batz, Mingshuai Chen, Sebastian Junges, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2023b. Probabilistic Program Verification via Inductive Synthesis of Inductive Invariants. In *TACAS (2) (Lecture Notes in Computer Science, Vol. 13994)*. Springer, 410–429. https://doi.org/10.1007/978-3-031-30820-8_25

Kevin Batz, Mingshuai Chen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Philipp Schröer. 2021a. Latticed k-Induction with an Application to Probabilistic Programs. In *CAV (2) (Lecture Notes in Computer Science, Vol. 12760)*. Springer, 524–549. https://doi.org/10.1007/978-3-030-81688-9_25

Kevin Batz, Adrian Gallus, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Tobias Winkler. 2022. Weighted programming: a programming paradigm for specifying mathematical models. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–30. https://doi.org/10.1145/3527310

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2021b. Relatively complete verification of probabilistic programs: an expressive language for expectation-based reasoning. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. https://doi.org/10.1145/3434320

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. 2019. Quantitative separation logic: a logic for reasoning about probabilistic pointer programs. *Proc. ACM Program. Lang.* 3, POPL (2019), 34:1–34:29. https://doi.org/10.1145/3290347

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Lena Verscht. 2023c. A Calculus for Amortized Expected Runtimes. *Proc. ACM Program. Lang.* 7, POPL (2023), 1957–1986. https://doi.org/10.1145/3571260

Noam Berger, Nevin Kapur, Leonard J. Schulman, and Vijay V. Vazirani. 2008. Solvency Games. In *FSTTCS (LIPIcs, Vol. 2)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 61–72. https://doi.org/10.4230/LIPICS.FSTTCS.2008.1741

David Blackwell. 1967. Positive dynamic programming. In *Proceedings of the 5th Berkeley symposium on Mathematical Statistics and Probability*, Vol. 1. University of California Press Berkeley, 415–418.

Craig Boutilier, Raymond Reiter, and Bob Price. 2001. Symbolic Dynamic Programming for First-Order MDPs. In *IJCAI*. Morgan Kaufmann, 690–700.

Tomás Brázdil, Václav Brozek, Kousha Etessami, Antonín Kucera, and Dominik Wojtczak. 2010. One-Counter Markov Decision Processes. In *SODA*. SIAM, 863–874. https://doi.org/10.1137/1.9781611973075.70

Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975), 453–457.

Klaus Dräger, Vojtech Forejt, Marta Z. Kwiatkowska, David Parker, and Mateusz Ujma. 2015. Permissive Controller Synthesis for Probabilistic Systems. *Log. Methods Comput. Sci.* 11, 2 (2015). https://doi.org/10.2168/LMCS-11(2:16)2015

Kousha Etessami and Mihalis Yannakakis. 2015. Recursive Markov Decision Processes and Recursive Stochastic Games. *J. ACM* 62, 2 (2015), 11:1–11:69. https://doi.org/10.1145/2699431

Lu Feng, Clemens Wiltsche, Laura R. Humphrey, and Ufuk Topcu. 2015. Controller synthesis for autonomous systems interacting with human operators. In *ICCPS*. ACM, 70–79. https://doi.org/10.1145/2735960.2735973

Yijun Feng, Lijun Zhang, David N. Jansen, Naijun Zhan, and Bican Xia. 2017. Finding Polynomial Loop Invariants for Probabilistic Programs. In *ATVA (Lecture Notes in Computer Science, Vol. 10482)*. Springer, 400–416. https://doi.org/10.1007/978-3-319-68167-2_26

Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. 2012. Operational Versus Weakest Precondition Semantics for the Probabilistic Guarded Command Language. In *QEST*. IEEE Computer Society, 168–177. https://doi.org/10.1109/QEST.2012.21

Sofie Haesaert, Nathalie Cauchi, and Alessandro Abate. 2017. Certified policy synthesis for general Markov decision processes: An application in building automation systems. *Perform. Evaluation* 117 (2017), 75–103. https://doi.org/10.1016/J.PEVA.2017.09.005

Marcel Hark, Benjamin Lucien Kaminski, Jürgen Giesl, and Joost-Pieter Katoen. 2020. Aiming low is harder: induction for lower bounds in probabilistic program verification. *Proc. ACM Program. Lang.* 4, POPL (2020), 37:1–37:28. https://doi.org/10.1145/3371105

Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. 2022. The probabilistic model checker Storm. *Int. J. Softw. Tools Technol. Transf.* 24, 4 (2022), 589–610. https://doi.org/10.1007/S10009-021-00633-Z

Thai Son Hoang, Zhendong Jin, Ken Robinson, Annabelle McIver, and Carroll Morgan. 2005. Development via Refinement in Probabilistic B - Foundation and Case Study. In *ZB (Lecture Notes in Computer Science, Vol. 3455)*. Springer, 355–373.

Kenneth E. Iverson. 1962. *A Programming Language.* John Wiley & Sons, Inc., USA.

Sebastian Junges, Joost-Pieter Katoen, Guillermo A. Pérez, and Tobias Winkler. 2021. The complexity of reachability in parametric Markov decision processes. *J. Comput. Syst. Sci.* 119 (2021), 183–210. https://doi.org/10.1016/J.JCSS.2021.02.006

Benjamin Lucien Kaminski. 2019. *Advanced weakest precondition calculi for probabilistic programs.* Ph. D. Dissertation. RWTH Aachen University, Germany.

Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2019. On the hardness of analyzing probabilistic programs. *Acta Informatica* 56, 3 (2019), 255–285. https://doi.org/10.1007/S00236-018-0321-1

Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2016. Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs. In *ESOP (Lecture Notes in Computer Science, Vol. 9632)*. Springer, 364–389. https://doi.org/10.1007/978-3-662-49498-1_15

Joost-Pieter Katoen. 2016. The Probabilistic Model Checking Landscape. In *LICS*. ACM, 31–45. https://doi.org/10.1145/2933575.2934574

Joost-Pieter Katoen, Annabelle McIver, Larissa Meinicke, and Carroll C. Morgan. 2010. Linear-Invariant Generation for Probabilistic Programs: - Automated Support for Proof-Based Methods. In *SAS (Lecture Notes in Computer Science, Vol. 6337)*. Springer, 390–406. https://doi.org/10.1007/978-3-642-15769-1_24

Mark Kattenbelt, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. 2009. Abstraction Refinement for Probabilistic Software. In *VMCAI (Lecture Notes in Computer Science, Vol. 5403)*. Springer, 182–197. https://doi.org/10.1007/978-3-540-93900-9_17

Mark Kattenbelt, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. 2010. A game-based abstraction-refinement framework for Markov decision processes. *Formal Methods Syst. Des.* 36, 3 (2010), 246–280. https://doi.org/10.1007/S10703-010-0097-6

Dexter Kozen. 1983. A Probabilistic PDL. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*. ACM, 291–297. https://doi.org/10.1145/800061.808758

Dexter Kozen. 1985. A Probabilistic PDL. *J. Comput. Syst. Sci.* 30, 2 (1985), 162–178. https://doi.org/10.1016/0022-0000(85)90012-1

Igor Kozine and Lev V. Utkin. 2002. Interval-Valued Finite Markov Chains. *Reliab. Comput.* 8, 2 (2002), 97–113. https://doi.org/10.1023/A:1014745904458

Marta Z. Kwiatkowska, Gethin Norman, and David Parker. 2002. PRISM: Probabilistic Symbolic Model Checker. In *Computer Performance Evaluation / TOOLS (Lecture Notes in Computer Science, Vol. 2324)*. Springer, 200–204. https://doi.org/10.1007/3-540-46029-2_13

Leslie Lamport. 2002. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley.

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR (Dakar) (Lecture Notes in Computer Science, Vol. 6355)*. Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

Konstantinos Mamouras. 2016. Synthesis of Strategies Using the Hoare Logic of Angelic and Demonic Nondeterminism. *Log. Methods Comput. Sci.* 12, 3 (2016). https://doi.org/10.2168/LMCS-12(3:6)2016

Annabelle McIver and Carroll Morgan. 2005. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer. https://doi.org/10.1007/b138392

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2017. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Dependable Software Systems Engineering*. NATO Science for Peace and Security Series - D: Information and Communication Security, Vol. 50. IOS Press, 104–125. https://doi.org/10.3233/978-1-61499-810-5-104

Marcelo Navarro and Federico Olmedo. 2022. Slicing of probabilistic programs based on specifications. *Sci. Comput. Program.* 220 (2022), 102822. https://doi.org/10.1016/J.SCICO.2022.102822

Donald Ornstein. 1969. On the existence of stationary optimal strategies. *Proc. Amer. Math. Soc.* 20, 2 (1969), 563–569.

David Park. 1969. Fixpoint Induction and Proofs of Program Properties. *Machine intelligence* 5 (1969).

Martin L. Puterman. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley. https://doi.org/10.1002/9780470316887

Scott Sanner, Karina Valdivia Delgado, and Leliane Nunes de Barros. 2011. Symbolic Dynamic Programming for Discrete and Continuous State MDPs. In *UAI*. AUAI Press, 643–652.

Philipp Schröer, Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2023. A Deductive Verification Infrastructure for Probabilistic Programs. 7, OOPSLA2, Article 294 (oct 2023), 31 pages. https://doi.org/10.1145/3622870

Martijn van Otterlo and Marco A. Wiering. 2012. Reinforcement Learning and Markov Decision Processes. In *Reinforcement Learning*. Adaptation, Learning, and Optimization, Vol. 12. Springer, 3–42. https://doi.org/10.1007/978-3-642-27645-3_1

Wikipedia. 2023. Nim — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Nim&oldid=1163491825. [Online; accessed 10-July-2023].

Received 2023-07-11; accepted 2023-11-07