Towards a Proof System for Probabilistic Dynamic Logic

Einar Broch Johnsen[1], Eduard Kamburjan[1], Raul Pardo[2], Erik Voogd[1], and Andrzej Wąsowski[2]

[1] Department of Informatics, University of Oslo, Oslo, Norway
`{einarj,eduard,erikvoogd}@ifi.uio.no`
[2] IT University of Copenhagen, Copenhagen, Denmark
`{raup,wasowski}@itu.dk`

Abstract. Whereas the semantics of probabilistic languages has been extensively studied, specification languages for their properties have received less attention—with the notable exception of recent and ongoing efforts by Joost-Pieter Katoen and collaborators. In this paper, we revisit probabilistic dynamic logic (pDL), a specification logic for programs in the probabilistic guarded command language (pGCL) of McIver and Morgan. Building on dynamic logic, pDL can express both first-order state properties and probabilistic reachability properties. In this paper, we report on work in progress towards a deductive proof system for pDL. This proof system, in line with verification systems for dynamic logic such as KeY, is based on forward reasoning by means of symbolic execution.

Keywords: Deductive verification · Probabilistic programs · Dynamic logic

1 Introduction

Joost-Pieter Katoen has pioneered techniques for the verification of probabilistic systems, including numerous contributions on model-checking algorithms (e.g., [2,4,16]), including tools such as the probabilistic model checker Storm [10,14], as well as proof systems for deductive verification (e.g., [5–7,11,19,22,28]). Whereas this line of work is rooted in Hoare logics and weakest precondition calculi, we here consider a specification language for probabilistic systems based on dynamic logic [13]: probabilistic dynamic logic (pDL for short) [25]. We believe it is interesting to study specification languages and deductive verification based on dynamic logic because dynamic logic is strictly more expressive than Hoare logic and weakest precondition calculi; in fact, both can be embedded in dynamic logic [12]. In contrast to these calculi, dynamic logics are closed under logical operators such as first-order connectives and quantifiers; for example, program equivalence, relative to state formulae φ and ψ, can be expressed by the dynamic logic formula $\varphi \to [s_1]\psi \iff \varphi \to [s_2]\psi$. Consequently, specification languages based on dynamic logic, including pDL, have classical model-theoretic semantics known from logics: a satisfaction semantics.

```
 1  prize := 0 ⊓ (prize := 1 ⊓ prize := 2);
 2  choice := 0 1/3⊕ (choice:=1 1/2⊕ choice:=2);
 3  if (prize = choice)
 4     open := (prize+1)%3 ⊓ open := (prize+2)%3;
 5  else
 6     open := (2*prize-choice)%3;
 7  if (switch)
 8     choice := (2*choice-open)%3
 9  else
10     skip
```

Fig. 1. The Monty Hall Program in pGCL (Monty_Hall).

This paper revisits pDL and its model-theoretic semantics, and its main contribution is a deductive verification system for pDL based on forward reasoning about pDL judgments, in contrast to the backwards reasoning used in weakest-precondition-based approaches. To this aim, we sketch a proof system for pDL based on symbolic execution rules that collect constraints about probabilities, and a prototype implementation of this proof system using Crowbar [17], a modular symbolic execution engine, and the SMT solver Z3 [23] to solve probabilistic constraints.

2 Motivating Example

As an example of a probabilistic program in pGCL, consider the *Monty Hall game*, in which a host presents three doors to a player. The first door contains a prize and the other doors are empty. The player needs to decide (or guess) the door behind which the prize is hidden. The game proceeds as follows. First, the location of the prize is non-deterministically selected by the host. Then, the player selects a door. The host opens an empty door that was not selected by the player, who is asked whether she would like to switch doors. We determine, using pDL, what option increases the chances of winning the prize (switching or not).

Figure 1 shows a pGCL program, Monty_Hall, modeling the behavior of host and player. The program contains four variables: prize (the door hiding the prize), choice (the door selected by the player), open (the door opened by the host), switch (a Boolean indicating whether the user switches door in the last step). Note that the variable switch is undefined in the program, and will encode the strategy of the player. Line 1 models the host's non-deterministic choice of the door for the prize. Line 2 models the player's choice of door (uniformly over the three doors). Lines 3–6 model the selection of the door to open, from the doors that were not selected by the player. Lines 7–10 model whether the player switches door or not. For simplicity, we use a slight shortcut to compute the door to open and to switch in Lines 6 and 8, respectively. Note that for $x, y \in \{0, 1, 2\}$, the expression $z = (2x - y)$ **mod** 3 simply returns $z \in \{0, 1, 2\}$ such that $z \neq x$

and $z \neq y$. Similarly, the expressions $y = (x + 1)$ **mod** $3, z = (x + 2)$ **mod** 3 in Line 4 ensure that $y \neq x$, $z \neq x$ and $y \neq z$. This shortcut computes the doors that the host may open when the player chose the door with the prize in Line 2.

An example of `pDL` specification of the Monty Hall game is the following formula:

$$\mathit{switch} = \mathit{true} \rightarrow [\texttt{Monty\_Hall}]_{\boldsymbol{p}}(\mathit{choice} = \mathit{prize}). \tag{1}$$

This formula expresses that if the player's strategy is to change door (i.e., the program's precondition is given by the state formula $\mathit{switch} = \mathit{true}$), then the probability of reaching a state characterized by the postcondition $\mathit{choice} = \mathit{prize}$ after successfully executing `Monty_Hall`, is $\boldsymbol{p}$. But what should be the value of $\boldsymbol{p}$? We will show in Section 6 that we can prove this specification for $\boldsymbol{p} = \min(\boldsymbol{p}_0, \boldsymbol{p}_1, \boldsymbol{p}_2)$ where each $\boldsymbol{p}_i$ is the probability for the different locations of the prize.

3 Preliminaries

We briefly introduce the programming language `pGCL` [21], but first we recall Markov Decision Processes [3, 26], which we use to define the semantics of `pGCL`.

3.1 Markov Decision Processes

Markov decision processes are computational structures that feature both probabilistic and nondeterministic choice. We start from the following definition (e.g., [3, 26]):

Definition 1 (Markov Decision Process). *A* Markov Decision Process *(MDP) is a tuple $M = (\mathit{State}, \mathit{Act}, \mathbf{P})$ where (i) State is a countable set of states, (ii) Act is a countable set of actions, and (iii) $\mathbf{P} : \mathit{State} \times \mathit{Act} \rightarrow \mathit{Dist}(\mathit{State})$ is a partial transition probability function.*

Let σ denote the states and a the actions of an MDP. A state σ is *final* if no further transitions are possible from it, i.e. $(\sigma, a) \notin \mathrm{dom}(\mathbf{P})$ for any a. A *path*, denoted $\overline{\sigma}$, is a sequence of states $\sigma_1, \ldots, \sigma_n$ such that σ_n is final and there are actions $a_1, \ldots, a_{n-1}$ such that $\mathbf{P}(\sigma_i, a_i)(\sigma_{i+1}) \geq 0$ for $1 \leq i < n$. Let $\mathrm{final}(\overline{\sigma})$ denote the final state of a path $\overline{\sigma}$. To resolve non-deterministic choice, a *positional policy* π maps states to actions, so $\pi : \mathit{State} \rightarrow \mathit{Act}$. Given a policy π, we define a transition relation $\xrightarrow{\cdot}_\pi \subseteq \mathit{State} \times [0, 1] \times \mathit{State}$ on states that resolves all the demonic choices in $\mathbf{P}$ and write

$$\sigma \xrightarrow{p_i}_\pi \sigma' \text{ iff } \mathbf{P}(\sigma, \pi(\sigma))(\sigma') = p_i.$$

Similarly, the reflexive and transitive closure of the transition relation, $\xrightarrow{p}{}^*_\pi \subseteq \mathit{State} \times [0, 1] \times \mathit{State}$, defines the probability of a path as

$$p = \Pr(\overline{\sigma}) = 1 \cdot p_1 \cdots p_n \quad \text{ where } \sigma_1 \xrightarrow{p_1}_\pi \cdots \xrightarrow{p_n}_\pi \sigma_n. \tag{2}$$

Thus, a path with no transitions consists of a single state σ, and $\Pr(\sigma) = 1$. Let $\mathrm{paths}_\pi(\sigma)$ denote the set of all paths with policy π from σ to final states.

An MDP may have an associated *reward function* $r : \mathit{State} \to [0, 1]$ that assigns a real value $r(\sigma)$ to any state $\sigma \in \mathit{State}$. (In this paper we assume that rewards are zero everywhere but in the final states.) We define the *expectation* of the reward starting in a state σ as the greatest lower bound on the expected value of the reward over all policies; so the real valued function defined as

$$\mathbf{E}_{\sigma}(r) = \inf_{\pi} \mathbb{E}_{\sigma,\pi}(r) = \inf_{\pi} \sum_{\overline{\sigma} \in \mathrm{paths}_{\pi}(\sigma)} \Pr(\overline{\sigma})\, r(\mathrm{final}(\overline{\sigma})) \ , \tag{3}$$

where $\mathbb{E}_{\sigma,\pi}(r)$ stands for the *expected value* of the random variable induced by the reward function under the given policy, known as the *expected reward*.

In this paper, we assume that MDPs (and the programs we derive them from) arrive at final states with probability 1 under all policies. This means that the logic pDL that we will be defining and interpreting over these MDPs can only talk about properties of almost surely terminating programs, so in general it cannot be used to reason about termination without adaptation.

3.2 pGCL: A Probabilistic Guarded Command Language

As programming language, we consider the probabilistic guarded command language (pGCL) of McIver and Morgan [21], a core language which features both probabilistic and non-deterministic choice. We briefly recall the syntax of pGCL and it semantics, formulated as a probabilistic transition system.

Syntax of pGCL. Let X be a set of program variables and $x \in X$, the syntax of pGCL is defined as follows:

$$\begin{aligned}
v \ &::= \mathit{true} \mid \mathit{false} \mid 0 \mid 1 \mid \ldots \\
e \ &::= v \mid x \mid \mathit{op}\ e \mid e\ \mathit{op}\ e \\
\mathit{op} &::= + \mid - \mid * \mid / \mid > \mid == \mid \geq \\
s \ &::= s \sqcap s \mid s\ {}_{e}{\oplus}\ s \mid s; s \mid \texttt{skip} \mid x := e \mid \texttt{if}\ e\ \{s\}\ \texttt{else}\ \{s\} \mid \texttt{while}\ e\ \{s\}
\end{aligned}$$

Statements s include the *non-deterministic (or demonic) choice* $s_1 \sqcap s_2$ between branches s_1 and s_2, and $s\ {}_{e}{\oplus}\ s'$ for the *probabilistic choice* between s and s'. A non-deterministic program $s_1 \sqcap s_2$ will arbitrarily select a branch for execution, whereas in a probabilistic program $s\ {}_{e}{\oplus}\ s'$, if the expression e evaluates to a value $p \in [0, 1]$ given the current values for the program variables, then s and s' have probability p and $1 - p$ of being selected, respectively. Binary operators and the remaining statements have the usual meaning.

Semantics of pGCL. The semantics of a pGCL program s can now be defined as an MDP $\mathcal{M}_s$. A state σ of $\mathcal{M}_s$ is a pair of a *valuation* and a *program*, so $\sigma = \langle \varepsilon, s \rangle$ where the valuation ε is a mapping from all the program variables in s to concrete values (we may omit the program from this pair, if it is unambiguous in the context). The state $\langle \varepsilon, s \rangle$ represents an *initial state* of the program s given

$$\textsc{(Assign)}\quad \frac{\varepsilon' = \varepsilon[x \mapsto \varepsilon(e)]}{\langle \varepsilon, x := e\rangle \xrightarrow{1}_{\pi} \langle \varepsilon', \texttt{skip}\rangle}
\qquad \textsc{(Composition1)}\quad \frac{\langle \varepsilon, s_2\rangle \xrightarrow{p}_{\pi} \langle \varepsilon', s\rangle}{\langle \varepsilon, \texttt{skip}; s_2\rangle \xrightarrow{p}_{\pi} \langle \varepsilon', s\rangle}
\qquad \textsc{(Composition2)}\quad \frac{\langle \varepsilon, s_1\rangle \xrightarrow{p}_{\pi} \langle \varepsilon', s\rangle}{\langle \varepsilon, s_1; s_2\rangle \xrightarrow{p}_{\pi} \langle \varepsilon', s; s_2\rangle}$$

$$\textsc{(DemChoice)}\quad \frac{i \in \{1,2\} \qquad \pi\langle \varepsilon, s_1 \sqcap s_2\rangle = s_i}{\langle \varepsilon, s_1 \sqcap s_2\rangle \xrightarrow{1}_{\pi} \langle \varepsilon', s_i\rangle}
\qquad \textsc{(If1)}\quad \frac{\varepsilon(e) = \mathit{true}}{\langle \varepsilon, \texttt{if}\ e\ \{s_1\}\ \texttt{else}\ \{s_2\}\rangle \xrightarrow{1}_{\pi} \langle \varepsilon, s_1\rangle}
\qquad \textsc{(If2)}\quad \frac{\varepsilon(e) = \mathit{false}}{\langle \varepsilon, \texttt{if}\ e\ \{s_1\}\ \texttt{else}\ \{s_2\}\rangle \xrightarrow{1}_{\pi} \langle \varepsilon, s_2\rangle}$$

$$\textsc{(ProbChoice1)}\quad \frac{\varepsilon(e) = p \quad 0 \le p \le 1}{\langle \varepsilon, s_1 \,{}_{e}\!\oplus s_2\rangle \xrightarrow{p}_{\pi} \langle \varepsilon, s_1\rangle}
\qquad \textsc{(While1)}\quad \frac{\varepsilon(e) = \mathit{true}}{\langle \varepsilon, \texttt{while}\ e\ \{s\}\rangle \xrightarrow{1}_{\pi} \langle \varepsilon, s; \texttt{while}\ e\ \{s\}\rangle}$$

$$\textsc{(ProbChoice2)}\quad \frac{\varepsilon(e) = p \quad 0 \le p \le 1}{\langle \varepsilon, s_1 \,{}_{e}\!\oplus s_2\rangle \xrightarrow{1-p}_{\pi} \langle \varepsilon, s_2\rangle}
\qquad \textsc{(While2)}\quad \frac{\varepsilon(e) = \mathit{false}}{\langle \varepsilon, \texttt{while}\ e\ \{s\}\rangle \xrightarrow{1}_{\pi} \langle \varepsilon, \texttt{skip}\rangle}$$

Fig. 2. An MDP-semantics for pGCL.

some initial valuation ε and the state $\langle \varepsilon, \texttt{skip}\rangle$ represents a *final state* in which the program has terminated with the valuation ε. For a concrete program, the *policy* π is a function that determines how non-deterministic choice is resolved for a given valuation of the program variables; i.e., $\pi\langle \varepsilon, s_1 \sqcap s_2\rangle = s_i$ for either $i = 1$ or $i = 2$. The rules defining the partial transition probability function for a given policy π are shown in Figure 2.

4 PDL: Probabilistic Dynamic Logic

The probabilistic dynamic logic pDL was introduced by Pardo *et al.* [25] as a specification language for probabilistic programs in pGCL. pDL builds on dynamic logic [13], a modal logic in which logical formulae with programs in the modalities can be used to express reachability properties. Our formulation of pDL here differs from our previous work [25] by incorporating *symbolic updates* [8], a technique for representing state change in forward symbolic execution that is well-known from the KeY verification system [1].

4.1 Syntax of pDL

Given sets X of program variables and L of logical variables disjoint from X, let ATF denote the well-formed atomic formulae built using constants, program and logical variables. For every $l \in L$, let $\operatorname{dom} l$ denote the domain of l. Let x range over X and t over well-formed terms, which for our purposes are pGCL expressions where also logical variables are allowed.

The formulae φ of probabilistic dynamic logic (pDL) are defined inductively as the smallest set generated by the following grammar.

$$\begin{array}{lll} \varphi & ::= & \mathsf{ATF} \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \forall l \cdot \varphi \mid [s]_{\boldsymbol{p}}\, \varphi \mid \{U\}\varphi \\ U & ::= & \mathsf{empty} \mid x \;\mapsto\; t \end{array}$$

where φ ranges over pDL formulae, $l \in L$ over logical variables, s is a pGCL program with variables in X, and $\boldsymbol{p}\colon \mathit{State} \to [0,1]$ is an expectation assigning values in $[0,1]$ to initial states of the program s. The logical operators $\to$, $\vee$ and $\exists$ are derived in terms of $\neg$, $\wedge$ and $\forall$ as usual. As usual, state formulae are pDL formulae without the box-modality; we denote state formulae by FOL.

A formula can be constructed by applying symbolic updates U to formulae φ; i.e., $\{U\}\varphi$ is a well-formed formula. Note that formulae that include symbolic updates are typically used in intermediate steps in a proof system based on forward symbolic execution; symbolic updates are typically not part of user-provided specifications. Symbolic updates are syntactic representations of term substitutions for program variables, which keep track of symbolic state changes within a proof branch. The empty update empty denotes no change, the update $x \;\mapsto\; t$ denotes a state change (or substitution [1]) where the program variable x has the value of t. The update application $\{U\}\varphi$ applies the update U to φ.

4.2 Semantics of pDL

We extend valuations to also map logical variables $l \in L$ to values in $\operatorname{dom} l$ and let $\varepsilon \models_{\mathsf{ATF}} \varphi$ denote standard satisfaction, expressing that $\varphi \in \mathsf{ATF}$ holds in valuation ε. From now on, we equate valuations and states, writing, for instance, $\varepsilon \in \mathit{State}$. Though states consist not only of valuations, but also of program locations, the locations are not relevant for interpreting pDL formulae.

First, we define the semantics of updates as a function from program state to program state. Let $\varepsilon(t)$ be the evaluation of a term t in a state ε. The updated states $[\![U]\!](\varepsilon)$ for a given substitution U and state ε are given by

$$\begin{aligned} [\![\mathsf{empty}]\!](\varepsilon) &= \varepsilon \\ [\![x \mapsto t]\!](\varepsilon) &= \varepsilon[x \mapsto \varepsilon(t)] \end{aligned}$$

We define satisfiability of *well-formed formulae* in pDL as follows:

Definition 2 (Satisfiability of pDL Formulae). *Let φ be a well-formed* pDL *formula, π range over policies, $l \in L$, $\boldsymbol{p} : \mathit{State} \to [0,1]$ be an expectation lower bound, and ε be a valuation defined for all variables mentioned in φ. The* satisfiability *of a formula φ in a model ε, denoted $\varepsilon \models \varphi$, is defined inductively* *as follows:*

$$\begin{array}{lll}
\varepsilon \models \varphi & \text{iff} & \varphi \in \mathsf{ATF} \quad \text{and} \quad \varepsilon \models_{\mathsf{ATF}} \varphi \\
\varepsilon \models \varphi_1 \wedge \varphi_2 & \text{iff} & \varepsilon \models \varphi_1 \quad \text{and} \quad \varepsilon \models \varphi_2 \\
\varepsilon \models \neg\varphi & \text{iff} & \text{not } \varepsilon \models \varphi \\
\varepsilon \models \forall l \cdot \varphi & \text{iff} & \varepsilon \models \varphi[l := v] \text{ for each } v \in \operatorname{dom} l \\
\varepsilon \models \{U\}\varphi & \text{iff} & [\![U]\!](\varepsilon) \models \varphi \\
\varepsilon \models [s]_{\boldsymbol{p}}\varphi & \text{iff} & \boldsymbol{p}(\varepsilon) \leq \mathbf{E}_\varepsilon[\![\varphi]\!] \text{ where the expectation is taken in } \mathcal{M}_s
\end{array}$$

For $\varphi \in \mathsf{ATF}$, $\models_{\mathsf{ATF}}$ can be used to check satisfaction just against the valuation of program variables since φ is well-formed. In the case of universal quantification, the substitution replaces logical variables with constants. The last case (p-box) is implicitly recursive, since the characteristic function $[\![\varphi]\!]$ refers to the satisfaction of φ in the final states of s. We use the characteristic function $[\![\varphi]\!]$ as the reward function on the final state of $\mathcal{M}_s$. In other words, the satisfaction of a p-box formula $[s]_{\boldsymbol{p}}\,\varphi$ captures a lower bound on the probability of φ holding after the program s. Consequently, pDL supports specification and reasoning about probabilistic reachability properties in almost surely terminating programs. We use $\models [s]_{\boldsymbol{p}}\,\varphi$ to denote that a formula is *valid*, *i.e.*, $\varepsilon \models [s]_{\boldsymbol{p}}\,\varphi$ for all valuations ε.

Proposition 1 (Properties of pDL [25]). Let s, s_1, s_2 be pGCL programs, φ a pDL formula, and ε a valuation.

(i) (termination) $\varepsilon \models [\texttt{skip}]_{\mathbf{1}}\varphi$ if and only if $\varepsilon \models \varphi$;
(ii) (inaction) $\varepsilon \models [s]_{\boldsymbol{p}}\varphi$ if and only if $\varepsilon \models [\texttt{skip};\ s]_{\boldsymbol{p}}\varphi$;
(iii) (assign) $\varepsilon \models [x := e;\ s]_{\boldsymbol{p}}\varphi$ if and only if $\varepsilon[x \mapsto \varepsilon(e)] \models [s]_{\boldsymbol{p}}\varphi$;
(iv) (universal lower bound) $\varepsilon \models [s]_{\mathbf{0}}\varphi$;
(v) (quantitative weakening) if $\varepsilon \models [s]_{\boldsymbol{p}}\varphi$ and $\boldsymbol{p}' \leq \boldsymbol{p}$ then $\varepsilon \models [s]_{\boldsymbol{p}'}\varphi$;
(vi) (demonic choice) $\varepsilon \models [s_1]_{\boldsymbol{p}}\varphi$ and $\varepsilon \models [s_2]_{\boldsymbol{p}}\varphi$ if and only if $\varepsilon \models [s_1 \sqcap s_2]_{\boldsymbol{p}}\varphi$;
(vii) (probabilistic choice) if $\varepsilon \models [s_1]_{\boldsymbol{p}_1}\varphi$ and $\varepsilon \models [s_2]_{\boldsymbol{p}_2}\varphi$ then $\varepsilon \models [s_1\ {}_e{\oplus}\ s_2]_{\boldsymbol{p}}\varphi$, where $\boldsymbol{p} = \varepsilon(e)\boldsymbol{p}_1 + (1 - \varepsilon(e))\boldsymbol{p}_2$;
(viii) (if true) if $\varepsilon \models e \wedge [s_1]_{\boldsymbol{p}}\varphi$ then $\varepsilon \models [\texttt{if}\ (e)\ \{s_1\}\ \texttt{else}\ \{s_2\}]_{\boldsymbol{p}}\varphi$;
(ix) (if false) if $\varepsilon \models \neg e \wedge [s_2]_{\boldsymbol{p}}\varphi$ then $\varepsilon \models [\texttt{if}\ (e)\ \{s_1\}\ \texttt{else}\ \{s_2\}]_{\boldsymbol{p}}\varphi$; and
(x) (loop unfold) $\varepsilon \models [\texttt{if}\ (e)\ \{s;\ \texttt{while}\ (e)\ \{s\}\}\ \texttt{else}\ \{\texttt{skip}\}]_{\boldsymbol{p}}\varphi$ if and only if $\varepsilon \models [\texttt{while}\ (e)\ \{s\}]_{\boldsymbol{p}}\varphi$.

Note that in the above, arithmetic operations are lifted point-wise when applied to expectation lower bounds (functions $\boldsymbol{p}$, $\boldsymbol{p}$').

5 A Proof System for pDL

Judgments. Let Γ be a set of formulae and φ a singular formula. We write a *judgment* $\Gamma \vdash \varphi$ to state that the formula $\bigwedge \Gamma \to \varphi$ is valid, i.e., that $\varepsilon \models \bigwedge \Gamma \to \varphi$ for every ε. When defining rules, we denote with $\mathcal{U}$ a nested sequence of update applications of any depth, for example the schematic formula $\mathcal{U}\varphi$ matches $\{U_1\}\{U_2\}\ldots\{U_n\}\varphi$ for some natural n and updates $U_1, \ldots, U_n$.

Probabilistic constraints. Our proof system is going to work with constraints on probabilities. Let Φ be a set of pDL-formulae, $\mathcal{U}$ a symbolic update and eq an (in)equality over probability variables. A pDL-constraint $\langle eq \rangle_{\mathcal{U}}^{\Phi}$ expresses that $\mathcal{U}(eq)$ must hold in any state represented by the symbolic update $\mathcal{U}$ such that $\mathcal{U}(\Phi)$. In the sequel, we introduce the simplifying assumption that that probabilistic expressions in pDL do not depend on state variables, in which case we can simplify constraints to simple (in)equalities, e.g., $\boldsymbol{p} \leq \mathbf{1}$.

A probabilistic dynamic logic calculus for pGCL. We now introduce the inference rules for dynamic logic formulae of the form $\Gamma \Rightarrow \mathcal{U}[s]_{\boldsymbol{p}}\, \varphi$, expressing that if Γ holds in the initial state of some program execution, the probability of reaching a state in which φ holds from a state described by the symbolic update $\mathcal{U}$ by executing s, is at least $\boldsymbol{p}$ (as before applied to the initial state). Thus, if the symbolic update $\mathcal{U}$ is empty, we are in the initial state and s is the entire program to be analyzed. The application of the inference rules creates a proof tree in which pDL-constraints are generated as side conditions to rule applications. In the end, the proof-tree can be closed if its pDL-constraints are satisfiable.

Figure 3 shows the inference rules for pGCL—we omit inference rules for FOL connectives as they are standard, see [8] for details. The rules in Figure 3 are the syntax-driven, aiming to eliminate the weakening rule. Instead weakening is implicitly applied in the rules that add a pDL-constraint. Let us first consider the rule for **skip**, to explain the rule format. The rules symbolically execute the first statement of the box-modality, generating a premise that must hold for the symbolic execution of the remaining program s. Note that SKIP does not add any probabilistic constraint. In the rules EMPTY1 and EMPTY0, the empty program is denoted by a **skip**-statement without a continuation s. In these rules, our aim is to check that the postcondition φ holds in the current state, represented by the symbolic update $\mathcal{U}$ with a given probability $\boldsymbol{p}$. If $\boldsymbol{p} \leq \mathbf{0}$, we leave the proof tree open in rule EMPTY0. If $\boldsymbol{p} \geq \mathbf{1}$, we close the tree if the formula holds in rule EMPTY1.

Branching is expressed through probabilistic and demonic choice. Rule DEMON-CHOICE captures the demonic choice by taking the worst case of the two branches, where $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ capture the probability of the postcondition φ holding for each of the branches. The pDL-constraint $\langle \boldsymbol{p} \leq \mathbf{min}(\boldsymbol{p}_1, \boldsymbol{p}_2) \rangle$ captures this worst-case assumption. Rule PROBCHOICE similarly captures the probabilistic choice, here the pDL-constrain combines the probability f selecting a branch with the probability of reaching the postcondition in that branch. Conditional branching over an **if**-statement is captured by the rule IF, which is standard and simply adds the condition for selecting each branch in the state captured by the symbolic update $\mathcal{U}$ to the precondition of each premise.

Loops are captured by rule LOOPUNROLL which simply unfolds a **while**-statement into a conditional statement.

Asserting $\Gamma \vdash \varphi$ expresses existence of a proof tree with root $\Gamma \vdash \varphi$ whose set of side conditions is satisfiable.

Theorem 1 (Soundness). *For all Γ and φ, if $\Gamma \vdash \varphi$ then $\models (\bigwedge \Gamma) \rightarrow \varphi$.*

Proof. By induction on the height of the proof tree that justifies the judgment $\Gamma \vdash \varphi$, with an analysis of the rule that justifies the root of the proof tree. All rules except IF are of the form where the premise is $\Gamma \vdash \varphi_1$ and the conclusion is $\Gamma \vdash \varphi_2$. In such cases, it suffices to prove that if φ_1 holds then φ_2 holds. Throughout the proof, we use $\mathcal{U} = \{U_1\}\{U_2\}\ldots\{U_n\}$ for some n and symbolic updates $U_1, \ldots, U_n$. Then, using Definition 2, $\varepsilon \models \mathcal{U}\varphi$ if and only if $(\llbracket U_n \rrbracket \circ \llbracket U_{n-1} \rrbracket \circ \cdots \circ \llbracket U_1 \rrbracket)(\varepsilon) \models \varphi$. We will write $\rho_{\mathcal{U}}$ for the state transformation $\llbracket U_n \rrbracket \circ \llbracket U_{n-1} \rrbracket \circ \cdots \circ \llbracket U_1 \rrbracket$

- For rule EMPTY1 $\varepsilon \models \mathcal{U}(\varphi)$ iff $\rho_{\mathcal{U}}(\varepsilon) \models \varphi$ iff $\rho_{\mathcal{U}}(\varepsilon) \models [\texttt{skip}]_{\mathbf{1}}\varphi$ iff $\varepsilon \models \mathcal{U}[\texttt{skip}]_{\mathbf{1}}\varphi$. Here, we use Prop. 1(i).
- For rule SKIP, $\varepsilon \models \mathcal{U}[s]_{\boldsymbol{p}}\varphi$ iff $\rho_{\mathcal{U}}(\varepsilon) \models [s]_{\boldsymbol{p}}\varphi$ iff $\rho_{\mathcal{U}}(\varepsilon) \models [\texttt{skip}; s]_{\boldsymbol{p}}\varphi$ iff $\varepsilon \models \mathcal{U}[\texttt{skip}; s]_{\boldsymbol{p}}\varphi$. Here, we use Prop. 1(ii).
- For rule ASSIGN, $\varepsilon \models \mathcal{U}\{x \mapsto e\}[s]_{\boldsymbol{p}}\varphi$ iff $\rho_{\mathcal{U}}(\varepsilon) \models \{x \mapsto e\}[s]_{\boldsymbol{p}}\varphi$. Now write $\varepsilon' := \rho_{\mathcal{U}}(\varepsilon)$. Then $\varepsilon' \models \{x \mapsto e\}[s]_{\boldsymbol{p}}\varphi$ iff $\varepsilon'[x \mapsto \varepsilon'(e)] \models [s]_{\boldsymbol{p}}\varphi$ iff $\varepsilon' \models [x := e; s]_p\varphi$, using Prop. 1(iii).
- For rule EMPTY0, $p = 0$ is a universal lower bound for any formula: $\rho_{\mathcal{U}}(\varepsilon) \models [\texttt{skip}]_{\mathbf{0}}\varphi$, by Prop. 1 (universal lower bound).
- For rule DEMONCHOICE, let ε' be arbitrary. By assumption, $\varepsilon' \models \mathcal{U}[s_1; s]_{\boldsymbol{p}_1}\varphi$ and $\varepsilon' \models \mathcal{U}[s_2; s]_{\boldsymbol{p}_2}\varphi$. That is, with $\varepsilon := \rho_{\mathcal{U}}(\varepsilon')$, $\varepsilon \models [s_1; s]_{\boldsymbol{p}_1}\varphi$ and $\varepsilon \models [s_2; s]_{\boldsymbol{p}_2}$. Put $\boldsymbol{p} := \min(\boldsymbol{p}_1, \boldsymbol{p}_2)$. By Prop. 1(vi), $\varepsilon \models [s_1; s \sqcap s_2; s]_{\boldsymbol{p}}\varphi$, meaning $\boldsymbol{p} \leq \mathbf{E}_\varepsilon\llbracket\varphi\rrbracket$, taken in $\mathcal{M}_{s'}$ where $s' = s_1; s \sqcap s_2; s$. Analyzing the rules in Figure 2 and using the definition of expectation, this means that both $\boldsymbol{p} \leq \mathbf{E}_\varepsilon\llbracket\varphi\rrbracket$ taken in $\mathcal{M}_{s_1;s}$ and $\boldsymbol{p} \leq \mathbf{E}_\varepsilon\llbracket\varphi\rrbracket$ taken in $\mathcal{M}_{s_2;s}$. Now assume for contradiction that $\boldsymbol{p} > \mathbf{E}_\varepsilon\llbracket\varphi\rrbracket$ taken in $\mathcal{M}_{(s_1 \sqcap s_2);s}$. Then there is a policy π such that $\boldsymbol{p} > \mathbb{E}_{\varepsilon,\pi}\llbracket\varphi\rrbracket$ in $\mathcal{M}_{(s_1 \sqcap s_2);s}$. Using Rules DEMCHOICE and COMPOSITION2, then, depending on the policy but without loss of generality, $\boldsymbol{p} > \mathbb{E}_{\varepsilon,\pi}\llbracket\varphi\rrbracket$ also in $\mathcal{M}_{s_1;s}$. But this contradicts what we claimed before, that $\boldsymbol{p} \leq \mathbf{E}_\varepsilon\llbracket\varphi\rrbracket$ in $\mathcal{M}_{s_1;s}$. Conclude that $\varepsilon \models [(s_1 \sqcap s_2); s]_{\boldsymbol{p}}\varphi$ and so $\varepsilon' \models \mathcal{U}[(s_1 \sqcap s_2); s]_{\boldsymbol{p}}\varphi$. By Prop. 1(v), this is true for any $\boldsymbol{p} \leq \min(\boldsymbol{p}_1, \boldsymbol{p}_2)$, so we are done.
- For rule PROBCHOICE, let ε' be arbitrary. By assumption, $\varepsilon' \models \mathcal{U}[s_1; s]_{\boldsymbol{p}_1}\varphi$ and $\varepsilon' \models \mathcal{U}[s_2; s]_{\boldsymbol{p}_2}\varphi$, meaning $\varepsilon \models [s_1; s]_{\boldsymbol{p}_1}\varphi$ and $\varepsilon \models [s_2; s]_{\boldsymbol{p}_2}\varphi$ with $\varepsilon = \rho_{\mathcal{U}}(\varepsilon')$. With $\boldsymbol{p} := e \cdot \boldsymbol{p}_1 + (1 - e) \cdot \boldsymbol{p}_2$, by Prop. 1(vii), $\varepsilon \models [s_1; s \; {}_e\!\oplus \; s_2; s]_{\boldsymbol{p}}\varphi$. This means that $\boldsymbol{p} \leq \mathbf{E}_\varepsilon\llbracket\varphi\rrbracket$ in the MDP of $s_1; s \; {}_e\!\oplus \; s_2; s$, and analyzing Rules PROBCHOICE1, PROBCHOICE2, and COMPOSITION2, we know that

$$\underbrace{\mathbf{E}_\varepsilon\llbracket\varphi\rrbracket}_{\text{in } s_1;s\,{}_e\oplus s_2;s} = \varepsilon(e) \cdot \underbrace{\mathbf{E}_\varepsilon\llbracket\varphi\rrbracket}_{\text{in } s_1;s} + (1 - \varepsilon(e)) \cdot \underbrace{\mathbf{E}_\varepsilon\llbracket\varphi\rrbracket}_{\text{in } s_2;s} = \underbrace{\mathbf{E}_\varepsilon\llbracket\varphi\rrbracket}_{\text{in } (s_1\,{}_e\oplus s_2);s}$$

  we conclude that also $\varepsilon \models [(s_1 \; {}_e\!\oplus \; s_2); s]_{\boldsymbol{p}}\varphi$, and hence, $\varepsilon' \models \mathcal{U}[(s_1 \; {}_e\!\oplus \; s_2); s]_{\boldsymbol{p}}\varphi$. The argument is generalizable to any $\boldsymbol{p} \leq e \cdot \boldsymbol{p}_1 + (1 - e) \cdot \boldsymbol{p}_2$ (Prop. 1(v)), so we are done.
- For rule IF, let ε' be arbitrary and assume the premises of the rule hold. If $\varepsilon' \not\models \Gamma$ there is nothing to prove. Otherwise, w.l.o.g., $\varepsilon' \models \mathcal{U}(e)$ and therefore $\varepsilon' \models \mathcal{U}[s_1; s]_{\boldsymbol{p}}\varphi$. Hence, writing $\varepsilon = \rho_{\mathcal{U}}(\varepsilon)$, we have $\varepsilon \models e \wedge [s_1; s]_{\boldsymbol{p}}\varphi$. By Prop. 1(viii), then, $\varepsilon \models [\texttt{if } (e)\{s_1; s\} \texttt{ else } \{s_2; s\}]_{\boldsymbol{p}}\varphi$. This means that $\boldsymbol{p} \leq \mathbf{E}_\varepsilon\llbracket\varphi\rrbracket$ in the MDP of $\texttt{if } (e)\{s_1; s\} \texttt{ else } \{s_2; s\}$. Using Rules IF1

$$\textsc{(skip)}\quad \frac{\Gamma \vdash \mathcal{U}[s]_{\boldsymbol{p}}\ \varphi}{\Gamma \vdash \mathcal{U}[\texttt{skip};\ s]_{\boldsymbol{p}}\ \varphi}$$

$$\textsc{(demonChoice)}\quad \frac{\Gamma \vdash \mathcal{U}[s_1; s\ ]_{\boldsymbol{p}_1}\ \varphi \qquad \Gamma \vdash \mathcal{U}[s_2;\ s]_{\boldsymbol{p}_2}\ \varphi}{\Gamma \vdash \mathcal{U}[(s_1 \sqcap s_2);\ s]_{\boldsymbol{p}}\ \varphi}\ \langle \boldsymbol{p} \le \mathbf{min}(\boldsymbol{p}_1, \boldsymbol{p}_2)\rangle$$

$$\textsc{(assign)}\quad \frac{\Gamma \vdash \mathcal{U}\{x \mapsto e\}[s]_{\boldsymbol{p}}\ \varphi}{\Gamma \vdash \mathcal{U}[x := e;\ s]_{\boldsymbol{p}}\ \varphi}$$

$$\textsc{(probChoice)}\quad \frac{\Gamma \vdash \mathcal{U}[s_1;\ s]_{\boldsymbol{p}_1}\ \varphi \qquad \Gamma \vdash \mathcal{U}[s_2;\ s]_{\boldsymbol{p}_2}\ \varphi}{\Gamma \vdash \mathcal{U}[s_1\ {}_e\!\oplus s_2;\ s]_{\boldsymbol{p}}\ \varphi}\ \langle \boldsymbol{p} \le e \cdot \boldsymbol{p}_1 + (1-e)\cdot \boldsymbol{p}_2\rangle$$

$$\textsc{(empty1)}\quad \frac{\Gamma \vdash \mathcal{U}(\varphi)}{\Gamma \vdash \mathcal{U}[\texttt{skip}]_{\boldsymbol{p}}\ \varphi}\ \langle \boldsymbol{p} \doteq 1\rangle$$

$$\textsc{(if)}\quad \frac{\Gamma, \mathcal{U}(e) \vdash \mathcal{U}[s_1;\ s]_{\boldsymbol{p}}\ \varphi \qquad \Gamma, \neg\mathcal{U}(e) \vdash \mathcal{U}[s_2;\ s]_{\boldsymbol{p}}\ \varphi}{\Gamma \vdash \mathcal{U}[\texttt{if}\ (e)\{s_1\}\ \texttt{else}\ \{s_2\};\ s]_{\boldsymbol{p}}\ \varphi}$$

$$\textsc{(empty0)}\quad \frac{}{\Gamma \vdash \mathcal{U}[\texttt{skip}]_{\boldsymbol{p}}\ \varphi}\ \langle \boldsymbol{p} \doteq 0\rangle$$

$$\textsc{(loopUnroll)}\quad \frac{\Gamma \vdash \mathcal{U}[\texttt{if}\ (e)\ \{s_b;\ \texttt{while}\ (e)\ \{s_b\};\ s\}\ \texttt{else}\ s\}]_{\boldsymbol{p}}\ \varphi}{\Gamma \vdash \mathcal{U}[\texttt{while}\ (e)\ \{s_b\};\ s]_{\boldsymbol{p}}\ \varphi}$$

Fig. 3. Symbolic execution rules.

and Composition2 and the fact that $\varepsilon \models e$, also $\boldsymbol{p} \le \mathbf{E}_\varepsilon[\![\varphi]\!]$ for $s_1; s$. Then, using Rule If1 and $\varepsilon \models e$ again, $\boldsymbol{p} \le \mathbf{E}_\varepsilon[\![\varphi]\!]$ in $(\texttt{if}\ (e)\{s_1\}\ \texttt{else}\ \{s_2\}); s$, so that $\varepsilon \models [(\texttt{if}\ (e)\{s_1\}\ \texttt{else}\ \{s_2\}); s]_{\boldsymbol{p}}\varphi$, and so $\varepsilon' \models \mathcal{U}[(\texttt{if}\ (e)\{s_1\}\ \texttt{else}\ \{s_2\}); s]_{\boldsymbol{p}}\varphi$.

– Finally, for rule loopUnroll, let ε' be arbitrary and let $\varepsilon = \rho_{\mathcal{U}}(\varepsilon')$. There are two cases to consider:
  - If $\varepsilon \models e$ (meaning $\varepsilon(e) = \text{true}$) then: $\varepsilon \models [\texttt{while}\ (e)\ \{s_b\};\ s]_{\boldsymbol{p}}\varphi$ iff $\varepsilon \models [s_b;\ \texttt{while}\ (e)\ \{s_b\};\ s]_{\boldsymbol{p}}\varphi$ (using Rules While1 and Composition2 and definition of expectation) iff $\varepsilon \models [\texttt{if}\ (e)\ \{s_b;\ \texttt{while}\ (e)\ \{s_b\};\ s\}\ \texttt{else}\ s]_{\boldsymbol{p}}\varphi$ (using Rules If1 and Composition2).
  - Otherwise, if $\varepsilon \not\models e$, similar reasoning with Rules While2, If2, and Composition1 show that $\varepsilon \models [\texttt{while}\ (e)\ \{s_b\};\ s]_{\boldsymbol{p}}\varphi$ if and only if $\varepsilon \models [\texttt{if}\ (e)\ \{s_b;\ \texttt{while}\ (e)\ \{s_b\};\ s\}\ \texttt{else}\ s]_{\boldsymbol{p}}\varphi$

  In either case, we have shown that $\varepsilon' \models \mathcal{U}[\texttt{while}\ (e)\ \{s_b\};\ s]_{\boldsymbol{p}}\varphi$ if and only if $\varepsilon' \models \mathcal{U}[\texttt{if}\ (e)\ \{s_b;\ \texttt{while}\ (e)\ \{s_b\};\ s\}\ \texttt{else}\ s]_{\boldsymbol{p}}\varphi$

Example. We illustrate the use of the inference rules in Fig. 3 by considering one of the branches of the proof tree for property Eq. (1) of the *Monty Hall* game (Sect. 2). In Sect. 6, we use our prototype implementation to automatically generate the complete proof. The property we are interested in proving is as follows (the code for `Monty_Hall` is in Fig. 1):

$$\mathit{switch} = \mathit{true} \vdash [\texttt{Monty\_Hall}]_{\boldsymbol{p}}(\mathit{choice} = \mathit{prize})$$

First, we apply the rule for non-deterministic choice. For convenience, we use $\varphi \triangleq (\mathit{choice} = \mathit{prize})$.

$$\frac{\begin{array}{c}\mathit{switch} = \mathit{true} \vdash [\texttt{prize:=0; ...}]_{\boldsymbol{p}_0}\varphi \\ \mathit{switch} = \mathit{true} \vdash [\texttt{prize:=1; ...}]_{\boldsymbol{p}_1}\varphi \\ \mathit{switch} = \mathit{true} \vdash [\texttt{prize:=2; ...}]_{\boldsymbol{p}_2}\varphi\end{array}}{\mathit{switch} = \mathit{true} \vdash [\texttt{Monty\_Hall}]_{\boldsymbol{p}}\varphi}\ \boldsymbol{p} \le \min(\boldsymbol{p}_0, \boldsymbol{p}_1, \boldsymbol{p}_2)$$

The rule adds 3 different premises, one for each path of the non-deterministic choice. In what follows, we focus on the proof branch for $\boldsymbol{p}_0$. The next program statement is an assignment, thus we apply the rule Assign.

$$\frac{\mathit{switch} = \mathit{true} \vdash \{\mathit{prize} \mapsto 0\}[\texttt{choice:=0}\ {}_{1/3}\!\oplus \texttt{(choice:=1}\ {}_{1/2}\!\oplus \texttt{choice:=2);...}]_{\boldsymbol{p}_0}\varphi}{\mathit{switch} = \mathit{true} \vdash [\texttt{prize:=0; ...}]_{\boldsymbol{p}_0}\varphi}$$

This rule simply added the update $\{\mathit{prize} \mapsto 0\}$. In the following, we use Γ and $\mathcal{U}$ to denote, at a point in the proof tree, the set of formulae in the precedent of a judgment and the set of updates, respectively. Next we apply the rule to resolve the probabilistic choice:

$$\frac{\begin{array}{c}\Gamma \vdash \mathcal{U}[\texttt{choice:=0; ...}]_{\boldsymbol{p}_{00}}\varphi \\ \Gamma \vdash \mathcal{U}[\texttt{(choice:=1}\ {}_{1/2}\!\oplus \texttt{choice:=2);...}]_{\boldsymbol{p}_{01}}\varphi\end{array}}{\Gamma \vdash \mathcal{U}[\texttt{choice:=0}\ {}_{1/3}\!\oplus \texttt{(choice:=1}\ {}_{1/2}\!\oplus \texttt{choice:=2);...}]_{\boldsymbol{p}_0}\varphi}\ \boldsymbol{p}_0 \le 1/3\boldsymbol{p}_{00} + 2/3\boldsymbol{p}_{01}$$

We continue the example with the proof tree for the $\boldsymbol{p}_{00}$ premise. We apply Assign so that we add the assignment as a symbolic update $\mathcal{U} = \{\mathit{prize} \mapsto 0, \mathit{choice} \mapsto 0\}$. Then, we apply the rule for if-statements:

$$\frac{\Gamma, \overbrace{\mathcal{U}(\texttt{prize = choice})}^{0=0} \vdash [s_0\texttt{;...}]_{\boldsymbol{p}_{00}}\varphi \quad \Gamma, \overbrace{\neg\mathcal{U}(\texttt{prize = choice})}^{0\neq 0} \vdash [s_1\texttt{;...}]_{\boldsymbol{p}_{00}}\varphi}{\Gamma \vdash [\textbf{if}\ \texttt{(prize = choice)}\ \{s_0\}\ \textbf{else}\ \{s_1\}\texttt{;...}]_{\boldsymbol{p}_{00}}\varphi}$$

The branch for the right premise above can be closed as we have derived false $(0 \neq 0)$. For the left premise, we apply the rule for resolving the non-deterministic choice $[\texttt{open := (prize+1)\%3} \sqcap \texttt{open := (prize+1)\%3; ...}]_{\boldsymbol{p}_{00}}$. This step generates a new constraint $\boldsymbol{p}_{00} \le \min(\boldsymbol{p}_{000}, \boldsymbol{p}_{001})$. We continue discussing the left branch (`open := (prize+1)%3`). In this case, we include a new update $\{\mathit{open} \mapsto 1\}$, as $\mathcal{U}((\mathit{prize} + 1)\%3) = (0 + 1)\%3 = 1$. Since we have $\mathit{switch} = \mathit{true}$, for the final if-statement (**if** `(switch) choice := (2*choice - open)%3` **else skip**), the right premise $(\neg\mathcal{U}(e))$ of If can be trivially closed, and the left premise produces a final assignment $\{\mathit{choice} \mapsto 2\}$—as $\mathcal{U}((2\mathit{choice} - \mathit{open})\%3) = (2 \cdot 0 - 1)\%3 = 2$. All in all, we get $\Gamma \vdash \{\mathit{prize} \mapsto 0, \mathit{open} \mapsto 1, \mathit{choice} \mapsto 2\}(\mathit{prize} = \mathit{choice})$. Since, given this update, the property does not hold, we close the branch by applying Empty0, i.e., $\boldsymbol{p}_{000} = 0$.

In summary, this proof branch has produced the following set of constraints: $\boldsymbol{p} \leq \min(\boldsymbol{p}_0, \boldsymbol{p}_1, \boldsymbol{p}_2)$, $\boldsymbol{p}_0 \leq 1/3\boldsymbol{p}_{00} + 2/3\boldsymbol{p}_{01}$, $\boldsymbol{p}_{00} \leq \min(\boldsymbol{p}_{000}, \boldsymbol{p}_{001})$ and $\boldsymbol{p}_{000} = 0$. Similar reasoning can be applied in the remaining branches, which will produce additional sets of constraints. In Sect. 6, we use Crowbar to automatically generate the complete proof and constraints. Furthermore, Crowbar outsources the set of constraints to an SMT solver to determine whether there are values for the different $\boldsymbol{p}_i$ that satisfy all constraints.

Discussion. The above proof system exploits the results of our earlier work [25] to build an automated deduction system. In this system, we have placed a rather simple instrument for reasoning about loops. We recall that a classic, non-probabilistic loop-invariant rule for partial correctness has the following structure (cast in a dynamic logic flavor [1]):

$$\frac{\Gamma \vdash \mathcal{U}\ I \quad \Gamma, \mathcal{U}'(\neg e \wedge I) \vdash \mathcal{U}'[s]\ \varphi \quad \Gamma, \mathcal{U}'(e \wedge I) \vdash \mathcal{U}'[s_b]\ I)}{\Gamma \vdash \mathcal{U}[\texttt{while}\ (e)\ \{s_b\};\ s]\ \varphi}\ (\text{loopInvariant})$$

where $\mathcal{U}'$ denotes some symbolic update such that the invariant holds (thus abstracting from the number of iterations of the loop). Using our syntax (and exploiting the probability lower bound of one to encode a qualitative box formula), this rule corresponds to:

$$\frac{\Gamma \vdash \mathcal{U}\ I \quad \Gamma, \mathcal{U}'(\neg e \wedge I) \vdash \mathcal{U}'[s]_{\mathbf{1}}\ \varphi \quad \Gamma, \mathcal{U}'(e \wedge I) \vdash \mathcal{U}'[s_b]_{\mathbf{1}}\ I)}{\Gamma \vdash \mathcal{U}[\texttt{while}\ (e)\ \{s_b\};\ s]_{\mathbf{1}}\ \varphi}\ (\text{loopQual})$$

The key aspect is to invent both an invariant and the substitution at termination, which (roughly) correspond to proposing a loop invariant in Hoare logics. This rule generalizes in a straightforward way to the case when the loop body itself is non-probabilistic. Then, it suffices to show that the expectation of the loop successor s is preserved by the loop, obtaining:

$$\frac{\Gamma \vdash \mathcal{U}\ I \quad \Gamma, \mathcal{U}'(\neg e \wedge I) \vdash \mathcal{U}'[s]_{\boldsymbol{p}}\ \varphi \quad \Gamma, \mathcal{U}'(e \wedge I) \vdash \mathcal{U}'[s_b]_{\mathbf{1}}\ I)}{\Gamma \vdash \mathcal{U}[\texttt{while}\ (e)\ \{s_b\};\ s]_{\boldsymbol{p}}\ \varphi}\ (\text{loopPreserving})$$

In general, weakest-precondition style rules can be encoded, and as our next step, we intend to experiment with existing rules from that space in actual proofs [18].

6 Implementation

We have implemented the above proof procedure[3] in Crowbar [17], a modular symbolic execution engine which uses the SMT-solver Z3 [23] under the hood to

[3] The sources for our prototype are available at https://github.com/Edkamb/crowbar-tool/tree/PDL.

```
[Spec: Ensures(choice == prize)]
[Spec: Prob(2/3)]
{
Int prize = -1; Int choice = -1; Int open = -1;
Int sw = 1;

[Spec: Demonic]
if( True ) { prize = 0; }
else { [Spec: Demonic] if( True ) { prize = 1; }
                       else { prize = 2; } }

[Spec: Prob(1/3)]
if( True ) { choice = 0; }
else { [Spec: Prob(1/2)] if( True ) { choice = 1; }
                         else { choice = 2; } }

if(prize==choice){
    [Spec: Demonic] if( True ) { open = (prize + 1 ) % 3; }
                    else { open = (prize + 2) % 3; }}
else {open = ((2 * prize) - choice) % 3;}

if(sw==1){choice = ((2 * choice) - open) % 3;} else { skip; }
}
```

Fig. 4. The Monty Hall game in the ABS encoding for Crowbar.

solve arithmetic constraints: Crowbar performs the symbolic execution and uses Z3 to discharge formulas without modalities. Crowbar was originally developed to experiment with deductive proof systems for the active object modeling language ABS [15]. For rapid prototyping, we have therefore used the Crowbar front-end for ABS and encoded pGCL into the main block of ABS programs; we use the general annotation format [27] for ABS programs to capture probabilistic aspects of pDL as annotations of ABS programs. The probabilistic constraints accumulated during the symbolic execution of the pGCL program are then passed to Z3 if the proof can otherwise be closed.

The main block itself is annotated with two *specification elements*: first the annotation `[Spec: Ensures(e)]` captures the post-condition and the annotation `[Spec: Prob(e)]` captures its probability. Furthermore, we use annotations of statements to encode probabilistic and demonic choice-operators of pGCL using an ABS branching statement. Demonic choice is expressed with a `[Spec: Demonic]` annotation on the branching statement, whose guard is then ignored. Probabilistic choice is similarly expressed with a `[Spec: Prob(e)]` annotation.

To illustrate the encoding of pGCL and pDL into the ABS representation of Crowbar, Figure 4 shows the encoding of the Monty Hall game introduced in Section 2. If `sw` is set to 1, denoting that the player always switches, the proof obligation can be discharged, as expected. If `sw` is set to 0, denoting that the player never switches, the proof attempt fails.

7 Related and Future Work

The symbolic execution proof system for probabilistic dynamic logic presented in this paper formulated in the style of KeY [1], and its symbolic execution proof system for sequential Java. We specifically use their technique for symbolic updates [8] in the formulation of our symbolic execution proof system. In contrast to their work, our deductive verification system addresses probabilistic programs by accumulating probabilistic constraints, which are then resolved using Z3.

Voogd *et al.* [29] developed a symbolic execution framework for probabilistic programs, building on Kozen's work on the semantics of probabilistic programs with random variables [20]. However, this work, inspired by de Boer and Bonsangue's work on formalized symbolic execution [9], does not extend into a deductive proof system as discussed in this paper. It further focuses on probabilistic programs with random variables rather than probabilistic and demonic choice as introduced in `pGCL`. In future work, we aim to integrate ideas from this paper into our reasoning framework; in particular, it would be interesting to add support for observe-statements. We refer to Voogd *et al.* [29] for a detailed discussion of how different semantics for probabilistic programs relate to symbolic execution.

Katoen *et al.* [28] developed a generic deductive proof system for probabilistic programs. Their work uses an encoding into a dedicated verification engine for probabilistic reasoning, called Caesar. In line with systems like Viper [24], they develop a intermediate representation language. To address probabilistic reasoning, their intermediate language is probabilistic and generates verification conditions for an SMT solver. Our work uses Crowbar, which is also a modular verification system. However, Caesar is tailored for weakest-precondition-style backwards reasoning, while our work targets forward reasoning by means of symbolic execution.

Our investigation shows that invariant based reasoning for loops is surprisingly subtle for probabilistic programs. Here, Joost-Pieter Katoen and his collaborators have again led the way (e.g., [22]). We hope some of his work on probabilistic loops can carry over to backwards reasoning in a weakest pre-expectation framework to forward reasoning in a symbolic execution setting, but it is an open question today—an open question that we aim to understand—exactly how forward and backwards reasoning about probabilistic loops relate.

8 Conclusion

This paper reports on work in progress towards a proof system for `pDL`, a probabilistic dynamic logic for specifying properties about probabilistic programs. A nice feature of `pDL` is that it is closed under logical operators such as first-order connectives and quantifiers, and that it has a model-theoretic semantics in terms of a satisfiability-relation. Our approach to a proof system for pDL uses forward reasoning by combining symbolic execution with a constraint solver. We have outlined a proof system for deductive verification based on symbolic execution which collects constraints about probabilities as side conditions. These are then forwarded to the constraint solver. For our prototype implementation, we have used Crowbar, a modular deductive verification engine based on symbolic execution, and the SMT-solver Z3 to solve constraints on probabilities.

We would like to end this paper by thanking Joost-Pieter Katoen not only for all his outstanding work on deductive verification of probabilistic programs, but also for interesting discussions on the relationship between backwards and forwards reasoning in this setting. A next step for us will be to understand if (and hopefully, how) some of the work conducted by Katoen and colleagues on reasoning about loops can adapted to a forward-reasoning framework.

Acknowledgments. This work was partly funded by the EU project SM4RTENANCE (grant no. 101123423). We are grateful to Asmae Heydari Tabar for discussions on symbolic execution and Crowbar.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (2016), https://doi.org/10.1007/978-3-319-49812-6
2. Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.-P.: Model-checking algorithms for continuous-time Markov chains. IEEE Trans. Software Eng. **29**(6), 524–541 (2003), https://doi.org/10.1109/TSE.2003.1205180
3. Baier, C., Katoen, J.-P.: Principles of model checking. MIT Press (2008)
4. Baier, C., Katoen, J.-P., Hermanns, H.: Approximate symbolic model checking of continuous-time Markov chains. In: Baeten, J.C.M., Mauw, S. (eds.) Proc. 10th International Conference on Concurrency Theory (CONCUR'99). Lecture Notes in Computer Science, vol. 1664, pp. 146–161. Springer (1999), https://doi.org/10.1007/3-540-48320-9_12
5. Batz, K., Biskup, T.J., Katoen, J.-P., Winkler, T.: Programmatic strategy synthesis: Resolving nondeterminism in probabilistic programs. Proc. ACM Program. Lang. **8**(POPL), 2792–2820 (2024), https://doi.org/10.1145/3632935
6. Batz, K., Chen, M., Junges, S., Kaminski, B.L., Katoen, J.-P., Matheja, C.: Probabilistic program verification via inductive synthesis of inductive invariants. In: Sankaranarayanan, S., Sharygina, N. (eds.) Proc. 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2023). Lecture Notes in Computer Science, vol. 13994, pp. 410–429. Springer (2023), https://doi.org/10.1007/978-3-031-30820-8_25
7. Batz, K., Kaminski, B.L., Katoen, J.-P., Matheja, C.: Relatively complete verification of probabilistic programs: an expressive language for expectation-based reasoning. Proc. ACM Program. Lang. **5**(POPL), 1–30 (2021), https://doi.org/10.1145/3434320
8. Beckert, B., Klebanov, V., Weiß, B.: Dynamic logic for Java. In: Ahrendt et al. [1], pp. 49–106, https://doi.org/10.1007/978-3-319-49812-6_3
9. de Boer, F.S., Bonsangue, M.: Symbolic execution formally explained. Formal Aspects of Computing **33**(4), 617–636 (2021), https://doi.org/10.1007/s00165-020-00527-y

10. Dehnert, C., Junges, S., Katoen, J.-P., Volk, M.: A Storm is coming: A modern probabilistic model checker. In: Majumdar, R., Kuncak, V. (eds.) Proc. 29th International Conference on Computer Aided Verification (CAV 2017). Lecture Notes in Computer Science, vol. 10427, pp. 592–600. Springer (2017), https://doi.org/10.1007/978-3-319-63390-9_31
11. Feng, S., Chen, M., Su, H., Kaminski, B.L., Katoen, J.-P., Zhan, N.: Lower bounds for possibly divergent probabilistic programs. Proc. ACM Program. Lang. **7**(OOPSLA1), 696–726 (2023), https://doi.org/10.1145/3586051
12. Hähnle, R.: Dijkstra's legacy on program verification. In: Apt, K.R., Hoare, T. (eds.) Edsger Wybe Dijkstra: His Life, Work, and Legacy, pp. 105–140. ACM / Morgan & Claypool (2022), https://doi.org/10.1145/3544585.3544593
13. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. Foundations of Computing, MIT Press (Oct 2000)
14. Hensel, C., Junges, S., Katoen, J.-P., Quatmann, T., Volk, M.: The probabilistic model checker Storm. Int. J. Softw. Tools Technol. Transf. **24**(4), 589–610 (2022), https://doi.org/10.1007/s10009-021-00633-z
15. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010). Lecture Notes in Computer Science, vol. 6957, pp. 142–164. Springer (2010), https://doi.org/10.1007/978-3-642-25271-6_8
16. Junges, S., Ábrahám, E., Hensel, C., Jansen, N., Katoen, J.-P., Quatmann, T., Volk, M.: Parameter synthesis for Markov models: covering the parameter space. Formal Methods Syst. Des. **62**(1), 181–259 (2024), https://doi.org/10.1007/s10703-023-00442-x
17. Kamburjan, E., Scaletta, M., Rollshausen, N.: Deductive verification of active objects with Crowbar. Sci. Comput. Program. **226**, 102928 (2023), https://doi.org/10.1016/j.scico.2023.102928
18. Kaminski, B.L.: Advanced weakest precondition calculi for probabilistic programs. Ph.D. thesis, RWTH Aachen University, Germany (2019), http://publications.rwth-aachen.de/record/755408
19. Kaminski, B.L., Katoen, J.-P., Matheja, C., Olmedo, F.: Weakest precondition reasoning for expected runtimes of randomized algorithms. J. ACM **65**(5), 30:1–30:68 (2018), https://doi.org/10.1145/3208102
20. Kozen, D.: Semantics of probabilistic programs. In: Proc. 20th Annual Symposium on Foundations of Computer Science. pp. 101–114. IEEE Computer Society (1979), https://doi.org/10.1109/SFCS.1979.38
21. McIver, A., Morgan, C.: Abstraction, Refinement and Proof for Probabilistic Systems. Monographs in Computer Science, Springer (2005), https://doi.org/10.1007/b138392
22. McIver, A., Morgan, C., Kaminski, B.L., Katoen, J.-P.: A new proof rule for almost-sure termination. Proc. ACM Program. Lang. **2**(POPL), 33:1–33:28 (2018), https://doi.org/10.1145/3158121
23. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Proc. 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008). Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008), https://doi.org/10.1007/978-3-540-78800-3_24
24. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) Proc. 17th International Conference on Verification, Model Checking, and Abstract

Interpretation (VMCAI 2016). Lecture Notes in Computer Science, vol. 9583, pp. 41–62. Springer (2016), https://doi.org/10.1007/978-3-662-49122-5_2

25. Pardo, R., Johnsen, E.B., Schaefer, I., Wąsowski, A.: A specification logic for programs in the probabilistic guarded command language. In: Proc. 19th International Colloquium on Theoretical Aspects of Computing (ICTAC'22). Lecture Notes in Computer Science, vol. 13572, pp. 369–387. Springer (2022), https://doi.org/10.1007/978-3-031-17715-6_24
26. Puterman, M.L.: Markov Decision Processes. Wiley (2005)
27. Schlatte, R., Johnsen, E.B., Kamburjan, E., Tapia Tarifa, S.L.: The ABS simulator toolchain. Sci. Comput. Program. **223**, 102861 (2022), https://doi.org/10.1016/j.scico.2022.102861
28. Schröer, P., Batz, K., Kaminski, B.L., Katoen, J.-P., Matheja, C.: A deductive verification infrastructure for probabilistic programs. Proc. ACM Program. Lang. **7**(OOPSLA2), 2052–2082 (2023), https://doi.org/10.1145/3622870
29. Voogd, E., Johnsen, E.B., Silva, A., Susag, Z.J., Wąsowski, A.: Symbolic semantics for probabilistic programs. In: Jansen, N., Tribastone, M. (eds.) Proc. 20th International Conference on Quantitative Evaluation of Systems (QEST 2023). Lecture Notes in Computer Science, vol. 14287, pp. 329–345. Springer (2023), https://doi.org/10.1007/978-3-031-43835-6_23