PRINSYS—on a Quest for Probabilistic Loop Invariants ⋆

Friedrich Gretz[1,2], Joost-Pieter Katoen[1], and Annabelle McIver[2]

[1] RWTH Aachen University, Germany
lastname@cs.rwth-aachen.de,
[2] Macquarie University, Australia
firstname.lastname@mq.edu.au

Abstract. PRINSYS (pronounced "princess") is a new software-tool for probabilistic invariant synthesis. In this paper we discuss its implementation and improvements of the methodology which was set out in previous work. In particular we have substantially simplified the method and generalised it to non-linear programs and invariants. PRINSYS follows a constraint-based approach. A given parameterised loop annotation is speculatively placed in the program. The tool returns a formula that captures precisely the invariant instances of the given candidate. Our approach is sound and complete. PRINSYS's applicability is evaluated on several examples. We believe the tool contributes to the successful analysis of sequential probabilistic programs with infinite-domain variables and parameters.

Keywords: invariant generation, probabilistic programs, non-linear constraint solving

1 Introduction

Motivation. Probabilistic programs are pivotal in different application fields like security, privacy [2]—several probabilistic protocols (e.g. onion-routing) aim to ensure privacy, and there is an increasing interest in the topic, partly driven by the social-media world—and cryptography [1] as well as quantum computing [13]. Such programs are single threaded and typically consist of a small number of code lines, but are hard to understand and analyse. The two major reasons for their complexity are the occurrence of program variables with unbounded domains, and parameters. Such parameters can be either loop bounds, number of participants (in a protocol), or probabilistic choices where the parameters range over concrete probabilities. For example, the following simple program generates a sample x according to a geometric distribution with parameter p. In every loop iteration, the variable x is increased by one with probability $1-p$

⋆ This work is partially funded by the DFG Research Training Group Algosyn, the EU FP7 Project CARP (Correct and Efficient Accelerator Programming), and the EU MEALS exchange project with Latin America.

Listing 1. $x \sim \text{geom}(p)$

```
x := 0;
flip := 0;
while (flip = 0) {
    ( flip := 1 [p] x := x+1 );
}
```

and *flip* is set to one with probability p, where p is an unknown real value from the range $(0, 1)$. The occurrence of unbounded variables and parameters comes at a price, namely that probabilistic programs in general cannot be analysed automatically by model-checking tools such as PRISM [10], PARAM [6], PASS [5] or APEX [9].

Approach. Instead we resort to deductive techniques. Recall that one of the main approaches to the verification of sequential programs rests on the pioneering work of Floyd, Hoare, and Dijkstra in which annotations are associated with control points in the program. Whereas the annotations for sequential programs are qualitative and can be expressed in predicate logic, quantitative annotations are needed to reason about probabilistic program correctness. McIver and Morgan [11] have extended the method of Floyd, Hoare, and Dijkstra to probabilistic programs by making the annotations real- rather than Boolean-valued expressions in the program variables. Using these methods we can prove that in the above program the average value of x is $\frac{1-p}{p}$. Annotating a probabilistic program with such expressions is non-trivial and undecidable in general. The main reason is the occurrence of loops. This all boils down to the question on how to establish a loop invariant. It is known that this is a notorious hard problem for traditional programs. For probabilistic programs it is even more difficult as loop invariants are quantitative—so-called probabilistic loop invariants. Variables do no longer have a value, but have a certain value with a given likelihood. Finding an invariant is hard and requires both ingenuity as well as involved computations to check that a given expression is indeed invariant. Recently, Katoen *et al.* [7] have proposed a technique for finding linear invariants for linear probabilistic programs. Linearity refers to the fact that right-hand sides of assignments and guards are linear expressions in the program variables (and parameters). This technique is based on speculatively annotating a loop with a template (in fact a linear inequality) and using constraint solving techniques to distill all parameters for which the template is indeed a loop invariant.

Contributions of this paper. The contributions of this paper are manifold. First and foremost, this paper presents PRINSYS (pronounce "princess"), a novel tool for supporting the semi-automated generation of probabilistic invariants

of pGCL[3] programs. This publicly available tool implements the technique advocated in [7], i.e., automatically computes the constraints under which a user-provided template is invariant, saving the user from tedious and error prone calculations. To the best of our knowledge, it is the first tool for synthesizing probabilistic invariants. Secondly, we show that the theory in [7] can be considerably simplified. In particular, we show that the usage of Motzkin's transposition theorem (a generalisation of Farkas' lemma) to turn an existentially quantified formula into a universally quantified one, is not needed. As a result, PRINSYS allows arbitrary formulas in templates and program guards. This allows for polynomial invariant templates and non-linear program expressions. So, an immediate consequence of this simplification is that the restriction to linear programs and linear invariants can be dropped. This is more of theoretical interest than of practical interest, as polynomial invariants—as for the traditional, non-probabilistic setting—are hard to synthesize in practice. Finally, we present some applications of the tool such as proving the equivalence of two programs computing a sample from $X-Y$ where X and Y are both geometrically distributed, and the generation of a fair coin from a biased one. We evaluate the experiments and give directions for future research.

Organization of the paper. Section 2 provides the preliminaries such as pGCL, probabilistic invariants, and expectations. Section 3 presents the steps of our approach and the simplification of [7]. Section 4 provides three examples to give insight about what PRINSYS can establish. Section 5 evaluates the tool and approach, whereas Sect. 6 concludes the paper and provides pointers to future work.

2 Background

When probabilistic programs are executed they determine a probability distribution over final values of program variables. For instance, on termination of

$$(x := 1 \ [0.75] \ x := 2);$$

the final value of x is 1 with probability $\frac{3}{4}$ or 2 with probability $1 - \frac{3}{4} = \frac{1}{4}$. An alternative way to characterise that probabilistic behaviour is to consider the expected values over random variables with respect to that distribution. For example, to determine the probability that x is set to 1, we can compute the expected value of the random variable "x is 1" which is $\frac{3}{4} \cdot 1 + \frac{1}{4} \cdot 0 = \frac{3}{4}$. Similarly, to determine the average value of x, we compute the expected value of the random variable "x" which is $\frac{3}{4} \cdot 1 + \frac{1}{4} \cdot 2 = \frac{5}{4}$. More generally, rather than a distribution-centred approach, we take an "expectation transformer" [11] approach. We annotate probabilistic programs with *expectations*.

[3] pGCL extends Dijkstra's guarded command language with a probabilistic choice operator.

Expectations. Expectations map program states to non-negative real values. They generalise Hoare's predicates for non-probabilistic programs towards real-valued functions. Intuitively, implication between predicates is generalised to pointwise inequality between expectations. For convenience we use square brackets to link Boolean truth values to numbers and by convention [true] = 1 and [false] = 0. In the example above, we call "x" the *post-expectation* and $\frac{5}{4}$ its *pre-expectation*. Thus the annotated program is $\langle \frac{5}{4} \rangle$ $(x := 1\ [0.75]\ x := 2)$; $\langle x \rangle$.

The formal mechanism for computing pre-expectations for a given program and post-expectation is the expectation transformer semantics [11]. Expectation transformers are the quantitative pendant to Dijkstra's predicate transformers. McIver and Morgan extend Dijkstra's concept and introduce a function wp(*prog*,*post*) which based on the program *prog* determines the *greatest pre-expectation* for any given post-expectation *post*. A summary of pGCL's expectation transformer semantics is given in Table 1 where f is a given post-expectation. From an operational perspective, pGCL programs can be viewed as (infinite state) MDPs with a reward structure induced by the given post-expectation f. Then the greatest pre-expectation can be computed as the expected cummulative reward on that model [4].

syntax *prog*	**semantics** wp(*prog*,f)
skip	f
abort	0
x := E	$f[x/E]$
P ; Q	$\mathrm{wp}(P, \mathrm{wp}(Q, f))$
if (G) { P } **else** { Q }	$[G] \cdot \mathrm{wp}(P, f) + [\neg G] \cdot \mathrm{wp}(Q, f)$
P [] Q	$\min\{\mathrm{wp}(P, f), \mathrm{wp}(Q, f)\}$
P [p] Q	$p \cdot \mathrm{wp}(P, f) + (1 - p) \cdot \mathrm{wp}(Q, f)$
while (G) { P }	$\mu X.([G] \cdot \mathrm{wp}(P, X) + [\neg G] \cdot f)$

Table 1. Syntax and expectation transformer semantics of pGCL

For loop-free programs, the pre-expectation is simply given by syntactic rules. However, loops pose a problem because their expectation over final values is given in terms of a least fixed point (over the domain of expectations with the ordering $\leq$, a pointwise ordering on expectations).

Invariants. Using special expectations which we call *invariants* we can avoid the calculation of a loop's fixed point. Assume we are given two expectations *pre* and *post* and we want to show that *pre* is a lower bound on the loop's actual pre-expectation, i.e.

$$pre \leq \mathrm{wp}(\mathbf{while}(G)\{body\}, post) \ .$$

Instead of computing the greatest pre-expectation $\mathrm{wp}(\mathbf{while}(G)\{\mathit{body}\}, \mathit{post})$ directly, it is more practical to divide this problem into simpler subtasks:

1. find an expectation $\mathcal{I}$ such that
$$\mathit{pre} \leq \mathcal{I} \quad \text{and} \quad \mathcal{I} \cdot [\neg G] \leq \mathit{post} \ ,$$
2. show $\mathcal{I}$ is *invariant*[4], that is $\mathcal{I} \cdot [G] \leq \mathrm{wlp}(\mathit{body}, \mathcal{I})$
3. show $\mathcal{I}$ is sound, that is $\mathcal{I} \leq \mathrm{wp}(\mathbf{while}(G)\{\mathit{body}\}, \mathcal{I} \cdot [\neg G])$

Points 2. and 3. may seem odd as they resemble the original problem of proving an inequality between an expectation and the greatest pre-expectation of a loop. However they are easier than the original problem, because in 2. the greatest pre-expectation can be explicitly computed because *body* is a loop-free program. In order to guarantee soundness (point 3.) the loop must terminate with probability one and the invariant $\mathcal{I}$ has to additionally meet one of the following sufficient conditions [11]:

- from every initial state of the loop only a finite state space is reachable
- or $\mathcal{I}$ is bounded above by some fixed constant
- or $\mathrm{wp}(\mathit{body}, \mathcal{I} \cdot [G])$ tends to zero as the number of iterations tends to infinity.

Remark 1. It is an open problem to give the *necessary and sufficient* conditions for soundness.

Put all together this proves the inequality above as

$$\mathit{pre} \leq \mathcal{I} \leq \mathrm{wp}(\mathbf{while}(G)\{\mathit{body}\}, \mathcal{I} \cdot [\neg G]) \leq^5 \mathrm{wp}(\mathbf{while}(G)\{\mathit{body}\}, \mathit{post}) \ .$$

Example 1 (Application of invariants.). Consider the program *prog* in Lst. 2. On each iteration of the loop it sets x to -1 with probability 0.15, to 0 with probability 0.5 and to 1 with probability 0.35. We would like to prove that the probability to terminate in a state where $x = 1$ is 0.7 or equivalently

$$\mathrm{wp}(\mathit{prog}, [x = 1]) = 0.7 \ .$$

Instead of computing the least fixed point of the loop wrt. post-expectation $[x = 1]$, we can show that $\mathcal{I} = [x = 0] \cdot 0.7 + [x = 1]$ is invariant. If the loop terminates, we can establish:

$$\begin{aligned} [\neg G] \cdot \mathcal{I} &= [x \neq 0] \cdot [x = 0] \cdot 0.7 + [x = 1] \\ &= [x = 1] \ . \end{aligned}$$

[4] wlp is the "liberal" version of wp. Both expectation transformers coincide for almost surely terminating programs. Since in this paper we do not consider nested loops, i.e. *body* is loop-free (and hence surely terminates), we do not discuss the theoretical differences between wp and wlp here.

[5] wp is monotonic in its second argument [11].

Listing 2. A simple loop

```
x := 0;
while (x=0) {
      (x := 0;) [0.5] { (x := -1 [0.3] x := 1); }
}
```

At the beginning of the program the initialisation of x transforms the invariant to:

$$\begin{aligned} \mathrm{wp}(x := 0, \mathcal{I}) &= [0 = 0] \cdot 0.7 + [0 = 1] \\ &= 0.7 \ . \end{aligned}$$

In this way we obtain the annotation

$$\langle 0.7 \rangle \ x := 0; \ \langle \mathcal{I} \rangle \ \textbf{while}(x = 0)\{\ldots\} \ \langle [x \neq 0] \cdot [\mathcal{I}] = [x = 1] \rangle$$

as desired. It is sound because the program obviously terminates with probability one and $\mathcal{I}$ is bounded.

The crucial point in determining a pre-expectation of a program is to discover the necessary loop invariants for each loop. Checking soundness and carrying out subsequent calculations for the other program constructs turns out be easy in practice. In the following section we explain our approach to finding invariants step by step.

3 Our Approach

To explain the steps carried out by PRINSYS we revisit the geometric distribution program from Lst. 1. In the next section, we will view it in a broader context.

Template. Consider the loop:

$$\textbf{while} \ (\mathrm{flip} \ = 0)\{ \ (\mathrm{flip} \ := 1 \ [\mathrm{p}] \ \mathrm{x} := \mathrm{x}{+}1); \ \}$$

and an expectation

$$\mathcal{T}_\alpha = [x \geq 0] \cdot x + [x \geq 0 \wedge \mathit{flip} = 0] \cdot \alpha$$

where α is an unknown (real) parameter. We call $\mathcal{T}_\alpha$ a *template*. Replacing α by a real value yields an *instance* of the template. Depending on this value, some instances may satisfy the invariance condition $\mathcal{T}_\alpha \cdot [G] \leq \mathrm{wlp}(\mathit{body}, \mathcal{T}_\alpha)$.

Goal. PRINSYS gives a characterisation of all invariant instances of a given template. This characterisation is a formula which is true for all admissible values of the template parameters, α in our example. It is important to stress that this method is *complete* in the sense that for any given template the resulting formula captures *precisely* the invariant instances.

Workflow. Stage 1: After parsing the program text and template, PRINSYS traverses the generated control flow graph of the program and computes:

$$\begin{aligned}&\mathrm{wp}(\mathit{flip} := 1\ [\mathrm{p}]\ \mathrm{x} := \mathrm{x}{+}1,\ \mathcal{T}_\alpha)\\ &= [x \geq 0] \cdot px + (1-p) \cdot ([x+1 \geq 0] \cdot (x+1) + [x+1 \geq 0 \wedge \mathit{flip} = 0] \cdot \alpha)\ .\end{aligned}$$

For details, cf. Table 1. After expanding this expression, the invariance condition amounts to:

$$\overbrace{[x \geq 0 \wedge \mathit{flip} = 0] \cdot (x+\alpha)}^{\mathcal{T}_\alpha \cdot [G]} \leq \begin{aligned}&[x \geq 0] \cdot px\\ &+ [x+1 \geq 0] \cdot ((1-p)x - p + 1)\\ &\underbrace{+[x+1 \geq 0 \wedge \mathit{flip} = 0] \cdot (1-p)\alpha}_{\mathrm{wlp}(\mathit{body}, \mathcal{T}_\alpha)}\ .\end{aligned}$$

Our goal is to find all α such that the point-wise inequality is satisfied, i.e. it holds for every x and every *flip*. This can be done by pairwise comparison of the summands on the left-hand side and the right-hand side. But summands may overlap. This makes it necessary to rewrite the expectations in disjoint normal form (DNF).

Theorem 1 (Transformation to DNF [7]). *Given an expectation of the form*

$$f = [P_1] \cdot w_1 + \ldots + [P_n] \cdot w_n.$$

Then an equivalent expectation in DNF can be written as:

$$\sum_{I \in \mathcal{P}(\overline{n}) \setminus \emptyset} \left(\left[\bigwedge_{i \in I} P_i \wedge \neg \left(\bigwedge_{j \in \mathcal{P}(\overline{n}) \setminus I} P_j \right) \right] \cdot \left(\sum_{i \in I} w_i \right) \right)$$

where $\overline{n}$ is the index set $\{1, \ldots, n\}$ and $\mathcal{P}(\cdot)$ denotes the power set.

The left-hand side of the inequality for the example program above is already in DNF as there is only one summand. We apply the transformation to the right-hand side expression. The result is an expectation with 15 summands. For better readability we only show the summands that are not trivially zero:

$$\begin{aligned}&[x+1 \geq 0 \wedge x < 0 \wedge \mathit{flip} = 0)] \cdot ((1-p)x + (1-p)\alpha - p + 1)\\ &+[x \geq 0 \wedge \mathit{flip} = 0)] \cdot (x + (1-p)\alpha - p + 1)\\ &+[x+1 \geq 0 \wedge x < 0 \wedge \mathit{flip} \neq 0] \cdot ((1-p)x - p + 1)\\ &+[x \geq 0 \wedge \mathit{flip} \neq 0] \cdot (x - p + 1)\,.\end{aligned}$$

The following theorem provides a straightforward encoding of the inequality as a first-order formula.

Theorem 2. *Given two expectations over variables $x_1, \ldots, x_n$ in disjoint-normal form*

$$f = [P_1] \cdot u_1 + \ldots + [P_M] \cdot u_M \ , \qquad g = [Q_1] \cdot w_1 + \ldots + [Q_K] \cdot w_K \ .$$

The inequality $f \leq g$ holds if and only if

$$\begin{aligned} \forall x_1, \ldots, x_n \in \mathbb{R} : & \bigwedge_{m \in \overline{M}} \bigwedge_{k \in \overline{K}} (P_m \wedge Q_k \Rightarrow (u_m - w_k \leq 0)) \\ & \wedge \bigwedge_{m \in \overline{M}} \left(P_m \wedge \left(\bigwedge_{k \in \overline{K}} \neg Q_k \right) \Rightarrow u_m \leq 0 \right) \\ & \wedge \bigwedge_{k \in \overline{K}} \left(Q_k \wedge \left(\bigwedge_{m \in \overline{M}} \neg P_m \right) \Rightarrow 0 \leq w_k \right) \end{aligned}$$

holds, where $\overline{X}$ is the set of indices $\{1, 2, \ldots, X\}$.

The idea is that we consider individual summands on the left-hand and right-hand side of the inequality and compare their values. It may also be the case that for some evaluations, all predicates on the right-hand side are false and hence the expectation is zero (i.e., the zero function). Then it must be ensured that no summand is greater than zero on the left-hand side. Conversely, if none of the predicates on the left-hand side are satisfied, the summands on the right-hand side may be no less than zero.

Theorem 2 originally appears in [7] where the last case is omitted because expectations are assumed to be non-negative by definition. However it is crucial to encode such informal assumptions in the formula as the tools are not aware of such expectation properties and instead treat them as usual functions over real values. This issue remained undiscovered until its implementation in Prinsys caused incorrect results. The lesson learned is that bridging the gap between an idea and a working implementation requires more than "just" coding.

Continuing our example, the (simplified) first-order formula obtained is:

$$\begin{aligned} \forall x, \mathit{flip} : & (\alpha p + p - 1 \leq 0 \vee \mathit{flip} \neq 0 \vee x < 0) \\ & \wedge (\alpha p - \alpha + px + p - x - 1 \leq 0 \vee \mathit{flip} \neq 0 \vee x + 1 < 0 \vee x \geq 0) \\ & \wedge (\mathit{flip} = 0 \vee px + p - x - 1 \leq 0 \vee x + 1 < 0 \vee x \geq 0) \\ & \wedge (\mathit{flip} = 0 \vee p - x - 1 \leq 0 \vee x < 0) \, . \end{aligned}$$

The calculation of this formula by Prinsys concludes the first stage.

Stage 2: The formula is passed to Redlog which simplifies the formula by quantifier elimination. Sometimes the result returned by Redlog still contains redundant information and can be further reduced by its built-in simplifiers or by the Slfq tool. In the end the user is presented a formula that characterises all αs that make $\mathcal{T}_\alpha$ invariant:

$$\alpha p + p - 1 \geq 0 \ .$$

Listing 3. Annotated program from Lst. 1

```
⟨(1−p)/p⟩
x := 0;
flip := 0;
⟨[x ≥ 0] · x + [x ≥ 0 ∧ flip = 0] · (1−p)/p⟩
while (flip = 0) {
    ( flip := 1 [p] x := x+1 );
}
⟨x⟩
```

We pick the greatest admissible α and obtain an invariant:

$$\mathcal{T}_{\frac{1-p}{p}} = [x \geq 0] \cdot x + [x \geq 0 \wedge \mathit{flip} = 0] \cdot \frac{1-p}{p} \quad .$$

This can be used to prove that the program in Lst. 1 has an average outcome of $\frac{1-p}{p}$ which indeed is the mean of a geometric distribution with parameter p. The annotated program now looks as follows: The soundness of our invariant is given because there is always a non-zero probability to exit the loop, cf. definition of invariants above.

Figure 1 pictures the described workflow of PRINSYS.

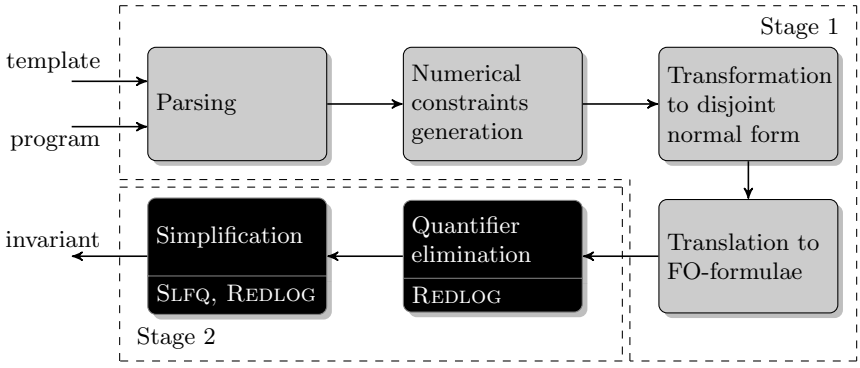

Fig. 1. Tool chain workflow

New insights. There are major differences with the approach sketched in [7]. In PRINSYS we skip the additional step of translating the universally quantified formula into an existential one using the Motzkin's transposition theorem. This step turns out to be not necessary. In fact it complicates matters as the existential formula will have more quantified variables which is bad for quantifier

Listing 4.

```
c := IC; // capital c (is set to some InitialCapital)
b := 1; // initially bet one unit
rounds := 0; //number of rounds played (survived)
while (b > 0){
        {// win with probability p
         c := c+b;
         b := 0;}
        [p]
        {// lose with probability 1-p
         c := c-b;
         b := 2*b;}
        rounds := rounds+1;
}
```

elimination. Furthermore, Motzkin's transposition theorem requires the universally quantified formula to be in a particular shape. Our implementation however does not have these restrictions and allows arbitrary predicates in the program's guards and in templates. Also the template and program do not have to be linear (theoretically at least) because Redlog and Slfq can work with polynomials. Moreover the invariant generation method remains complete in this case. This is because starting with the invariance condition all subsequent steps to obtain the simplified first-order formula are equivalence transformations.

This section has not only illustrated how the tool-chain works but also clearly shows the great amount of calculations that are done automatically for the user. Within seconds the user may try out different templates and play with the parameters until an invariant is found. The Prinsys tool saves the user a lot of tedious, error-prone work and pushes forward the automation of probabilistic program analysis.

4 Applications

This section presents three examples, for simplicity all based on our running example of the geometric distribution, that illustrate the possibilities of the Prinsys approach. Let us start with a relatively simple example.

Martingale betting strategy. Another variant of the geometric distribution appears in the following program, which models a gambler with infinite resources who is playing according to the martingale strategy. Note that this program has two unbounded variables. Using the same template as before, we discover that $\frac{1}{p}$ is the expected number of rounds played before the gambler stops. The expecta-

Listing 5.

```
x := 0;
flip := 0;
while (flip = 0) {
   (x := x+1 [p] flip := 1);
}
flip := 0;
while (flip = 0) {
   (x := x-1 [q] flip := 1);
}
```

Listing 6.

```
x := 0;
(flip := 0 [0.5] flip := 1);
if (flip = 0) {
   while (flip = 0) {
     (x := x+1 [p] flip := 1);
   }
} else {
   flip := 0;
   while (flip = 0) {
     x := x-1;
     (skip [q] flip := 1);
   }
}
```

tion differs from what we have computed for the program in Lst. 1 because here the counter is increased also on the last iteration before the loop terminates.

Geometric distribution. This example is taken from [8] where amongst others it has been shown that the two programs in Lst. 5 and Lst. 6 are equivalent for $p = \frac{1}{2}$ and $q = \frac{2}{3}$. The proof in [8] relies on language equivalence checking of probabilistic automata. Here, we show how the techniques supported by PRINSYS can be used to show that both programs are equivalent for any p and q satisfying $q = \frac{1}{2-p}$. Let us explain the example in more detail. The aim is to generate a sample x according to the distribution $X-Y$ where X is geometrically distributed with parameter $1-p$ and Y is geometrically distributed with $1-q$. Although it is not common to say that a distribution has a parameter $1-p$, it is natural in the context of these programs where x is manipulated with probability p and the loop is terminated with the remaining probability. The difference between the programs in Lst. 5 and Lst. 6 is that the first uses two loops in sequence whereas the latter needs only one out of two loops. Our goal is to determine when the two programs are equivalent, in the sense that they compute the same value for x on average.

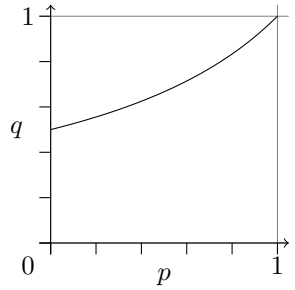

Fig. 2. Pairs (p, q) for which the programs in List. 5 and List. 6 produce the same x on average.

The PRINSYS tool generates invariants for single loops, so we consider each loop separately. Using the template $\mathcal{T}_\alpha = [x \geq 0] \cdot x + [x \geq 0 \wedge \mathit{flip} = 0] \cdot \alpha$ from our running example, PRINSYS yields the following invariants:

Listing 7. x is set to zero or one, each with probability 0.5

```
x := 0; // stores outcome of first biased coin flip
y := 0; // stores outcome of second biased coin flip

while (x-y = 0) {
    (x := 0 [p] x := 1);
    (y := 0 [p] y := 1);
}
```

- $\mathcal{I}_{11} = x + [\text{flip} = 0] \cdot \frac{p}{1-p}$,
- $\mathcal{I}_{12} = x + [\text{flip} = 0] \cdot \left(-\frac{q}{1-q}\right)$,
- $\mathcal{I}_{21} = \mathcal{I}_{11}$ and
- $\mathcal{I}_{22} = x + [\text{flip} = 0] \cdot \left(-\frac{1}{1-q}\right)$,

where $\mathcal{I}_{ij}$ is the invariant of the j-th loop in program i, $i, j \in \{1, 2\}$. With these invariants we can easily derive the expected value of x, which is $\frac{p}{1-p} - \frac{q}{1-q}$ and $\frac{p}{2(1-p)} - \frac{1}{2(1-q)}$ for the program in List. 5 and List. 6, respectively. The two programs thus are equivalent whenever these two expectations coincide; e. g. this is the case for $p = \frac{1}{2}$ and $q = \frac{2}{3}$ as discussed in [8]. Figure 2 visualises our result: for every point (p, q) on the graph the two programs are equivalent. This result cannot be obtained using the techniques in [8]; to the best of our knowledge there are no other automated techniques that can establish this.

Generating a fair coin from a biased coin. In [7], Hurd's algorithm to generate a sample according to a biased coin flip using only fair coin flips has been analysed. Using Prinsys the calculations can be automated. This was elaborated in [3]. Here we consider an algorithm for the opposite problem. Using a coin with some arbitrary bias $0 < p < 1$, the algorithm in Lst. 7 generates a sample according to a fair coin flip. The loop terminates when the biased coin was flipped twice and showed different outcomes. Obviously the program terminates with probability one as on each iteration of the loop there is a constant positive chance to terminate. The value of x is taken as the outcome. The two possible outcomes are characterised by $x = 0 \wedge y = 1$ and $x = 1 \wedge y = 0$. We encode these two possibilities in the template:

$$[x = 0 \wedge y - 1 = 0] \cdot (\alpha) + [x - 1 = 0 \wedge y = 0] \cdot (\beta)$$

Prinsys returns one constraint:

$$\alpha p^2 - \alpha p + \beta p^2 - \beta p \leq 0$$

As before we look for the maximum value, hence we consider equality with zero. The equation simplifies to $\alpha = -\beta$ because we know that $0 < p < 1$. Hence

$[x = 0 \wedge y - 1 = 0] - [x - 1 = 0 \wedge y = 0]$ is invariant[6] which, together with almost sure termination, gives us

$$\begin{aligned} & \mathrm{wp}(\mathit{prog}, [x = 0 \wedge y - 1 = 0] - [x - 1 = 0 \wedge y = 0]) \\ = \; & \mathrm{wp}(\mathit{prog}, [x = 0 \wedge y - 1 = 0]) - \mathrm{wp}(\mathit{prog}, [x - 1 = 0 \wedge y = 0]) \\ = \; & 0 \ . \end{aligned} \tag{1}$$

where *prog* is the entire program from Lst 7. The previous argument about almost sure termination and possible outcomes shows that

$$\begin{aligned} & \mathrm{wp}(\mathit{prog}, [x = 0 \wedge y - 1 = 0] + [x - 1 = 0 \wedge y = 0]) \\ = \; & \mathrm{wp}(\mathit{prog}, [x = 0 \wedge y - 1 = 0]) + \mathrm{wp}(\mathit{prog}, [x - 1 = 0 \wedge y = 0]) \\ = \; & 1 \ . \end{aligned} \tag{2}$$

The unique solution to (1) and (2) is

$$\begin{aligned} & \mathrm{wp}(\mathit{prog}, [x = 0 \wedge y - 1 = 0]) \\ = \; & \mathrm{wp}(\mathit{prog}, [x - 1 = 0 \wedge y = 0]) \\ = \; & 0.5 \ . \end{aligned}$$

This concludes the proof that x is distributed evenly for any p satisfying $0 < p < 1$.

5 Evaluation

We have seen three pGCL programs that were variants of the geometric distribution. Our approach allows us to exploit their common structure and enables us to calculate the expectation of these programs using the same template although they compute different (mean) values. Since our method does not rely on numerical calculation we are able to parameterise the programs and provide very general results. In particular we could decide when two programs have the same expectation depending on their parametric distributions. Another handy feature of reasoning with expectation-transformer wp is that we can exploit its properties as well. For example, the reasoning is modular with respect to sequential composition. That means we can compute the pre-expectation for individual loops and then add the results when we put the loops in sequence. The last example demonstrates yet another use of invariants. Instead of deriving a bound on the pre-expectation we have shown how an invariant may give constraints on the pre-expectation. Together with termination these constraints produced the sought pre-expectation. This exemplifies that invariants are not just a particular way to compute an expectation but rather they describe the behaviour of the program and can be used in different ways.

[6] We pick $\alpha = 1$ and $\beta = -1$ but in fact any non-zero pair of values $\alpha = -\beta$ would result in the same argument.

Together with the three (other) examples discussed in [4,7] we have a set of interesting programs which we can analyse with the help of PRINSYS. Note, that our examples do not make use of the non-deterministic choice statement. This is because the algorithms we focused on do not need it, however PRINSYS also supports non-deterministic pGCL programs. There is no commonly accepted benchmark suite that we can compare against as this area of research has not spawned many tools yet. We refrain from giving a table that shows for each program the state space size, the number of discovered invariants or running times. This is because the beauty of this approach is exactly that the number of states does not matter. In fact all programs that generate (a variant) of the geometric distribution have an infinite set of reachable states! The number of discovered invariants cannot be really be given as, first of all the result depends on the template provided and second we get a characterisation of *all* invariant instances of a template. Since we reason over the reals there are uncountably many.

The runtime of PRINSYS depends on the size of the expressions that we have to handle. This means that if we have many choices in the loop (i.e. there are many paths in the control flow graph) this will blow up the size of $\mathrm{wp}(\mathit{body}, \mathcal{T})$. The same is true for templates that have many summands. Finally, the external tools used by PRINSYS affect the overall running time. Their execution time cannot be predicted exactly but experience shows that the final simplification step takes considerably longer the more parameters we allow in the template. The overall runtime for the presented examples lies within a second on a laptop computer.

Since there is no software that could be easily adapted to support our methods, PRINSYS was developed from scratch. It was recently redesigned to be more extensible and easier to maintain as we hope that future developments in the area of constraint-based methods will use our work as a basis. From the user's point of view, the usability was substantially increased with the introduction of a graphical user interface that allows an intuitive interaction.

Programs and templates considered in our examples are linear. This means all guards, assignments or terms are linear in the program variables. As pointed out earlier, our approach per se allows polynomial expressions as well. To see to what extent this applies in practice we have tried to generate polynomial invariants for variants of a bounded random walk, cf. Lst. 8. The goal is here to estimate the number of steps taken before x hits its lower bound zero or upper bound M where M is a fixed parameter. Surprisingly quantifier elimination works reasonably fast for formulas with polynomials but the returned quantifier-free formula is very big. The lack of powerful simplification methods makes it difficult to find a concise representation of the formula that describes all invariant instances of the template. REDLOG's simplifier might increase the formula size or not terminate at all, whereas SLFQ hits the memory bound quickly and crashes, even if the allocated memory is increased maximally.

Listing 8. Bounded random walk

```
counter := 0;
while (x > 0 and x-M < 0){
    (x := x+1 [p] x := x-1);
    counter := counter+1;
}
```

6 Conclusion

We have presented a new software tool called PRINSYS for probabilistic invariant generation. Its functionality was explained and its merits were assessed in the discussion. Also implementation details that deviate from the theoretic description of the method in [7] were pointed out. During our evaluation we have reached the next challenge, that is to extend invariant generation to polynomial templates. Related work, e.g. [12] suggests a workaround to find polynomial invariants for non-probabilistic programs. This comes at the price that they sacrifice completeness and limit the class of systems permitted. In the future we would like to work out a similar approximate invariant generation method for probabilistic systems and evaluate it within PRINSYS.

References

1. Barthe, G., Grégoire, B., Béguelin, S.Z.: Probabilistic relational Hoare logics for computer-aided security proofs. In: Int. Conf. on Mathematics of Program Construction (MPC). LNCS, vol. 7342, pp. 1–6 (2012)
2. Barthe, G., Köpf, B., Olmedo, F., Béguelin, S.Z.: Probabilistic relational reasoning for differential privacy. In: Symp. on Principles of Programming Languages (POPL). pp. 97–110. ACM (2012)
3. Gretz, F.: Invariant Generation for Linear Probabilistic Programs. Master's thesis, RWTH Aachen (2010), `http://www-i2.informatik.rwth-aachen.de/i2/gretz/`
4. Gretz, F., Katoen, J.P., McIver, A.: Operational versus Weakest Precondition Semantics for the Probabilistic Guarded Command Language. In: QEST. pp. 168–177 (2012)
5. Hahn, E.M., Hermanns, H., Wachter, B., Zhang, L.: Pass: Abstraction refinement for infinite probabilistic models. In: TACAS. pp. 353–357 (2010)
6. Hahn, E.M., Hermanns, H., Zhang, L.: Probabilistic Reachability for Parametric Markov Models. STTT 13(1), 3–19 (2011)
7. Katoen, J.P., McIver, A., Meinicke, L., Morgan, C.: Linear-Invariant Generation for Probabilistic Programs. In: SAS, pp. 390–406. LNCS (2011)
8. Kiefer, S., Murawski, A., Ouaknine, J., Wachter, B., Worrell, J.: On the Complexity of the Equivalence Problem for Probabilistic Automata. In: FoSSaCS. LNCS, vol. 7213, pp. 467–481 (2012)
9. Kiefer, S., Murawski, A.S., Ouaknine, J., Wachter, B., Worrell, J.: APEX: An Analyzer for Open Probabilistic Programs. In: CAV. LNCS, vol. 7358, pp. 693–698 (2012)

10. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-time Systems. In: CAV. LNCS, vol. 6806, pp. 585–591 (2011)
11. McIver, A., Morgan, C.: Abstraction, Refinement And Proof For Probabilistic Systems (Monographs in Computer Science). SpringerVerlag (2004)
12. Sankaranarayanan, S., Sipma, H., Manna, Z.: Non-linear Loop Invariant Generation Using Gröbner Bases. In: POPL. pp. 318–329 (2004)
13. Ying, M.: Floyd-Hoare logic for quantum programs. ACM Trans. Program. Lang. Syst. 33(6), 19 (2011)