

Automated Sensitivity Analysis for Probabilistic Loops

Marcel Moosbrugger ¹, Julian Müllner ¹, and Laura Kovács ¹

TU Wien, Vienna, Austria

Abstract. We present an exact approach to analyze and quantify the sensitivity of higher moments of probabilistic loops with symbolic parameters, polynomial arithmetic and potentially uncountable state spaces. Our approach integrates methods from symbolic computation, probability theory, and static analysis in order to automatically capture sensitivity information about probabilistic loops. Sensitivity information allows us to formally establish how value distributions of probabilistic loop variables influence the functional behavior of loops, which can in particular be helpful when choosing values of loop variables in order to ensure efficient/expected computations. Our work uses algebraic techniques to model higher moments of loop variables via linear recurrence equations and introduce the notion of *sensitivity recurrences*. We show that sensitivity recurrences precisely model loop sensitivities, even in cases where the moments of loop variables do not satisfy a system of linear recurrences. As such, we enlarge the class of probabilistic loops for which sensitivity analysis was so far feasible. We demonstrate the success of our approach while analyzing the sensitivities of probabilistic loops.

Keywords: Probabilistic Programs · Sensitivity Analysis · Recurrences

1 Introduction

Probabilistic programs are imperative programs enriched with the capability to draw from probability distributions. By supporting native primitives to model uncertainty, probabilistic programming provides a powerful framework to model stochastic systems from many different areas, such as machine learning [19], biology [2], cyber-physical systems [15,30], cryptography [5], privacy [7], and randomized algorithms [27].

A challenging task in the analysis of probabilistic programs comes from the fact that values, or even value distributions, of symbolic parameters used within program expressions over probabilistic program variables are often unknown. Sensitivity analysis aims to quantify how small changes in such parameters influence computation results [1,4]. Sensitivity analysis thus provides additional information about the probabilistic program executions, even if some parameters are (partially) unknown. This sensitivity information can further be used, among others, in code optimization: sensitivity information quantifies the influence of parameters on the program variables, allowing to derive cost-effective estimates and optimize expected runtimes of probabilistic loops.

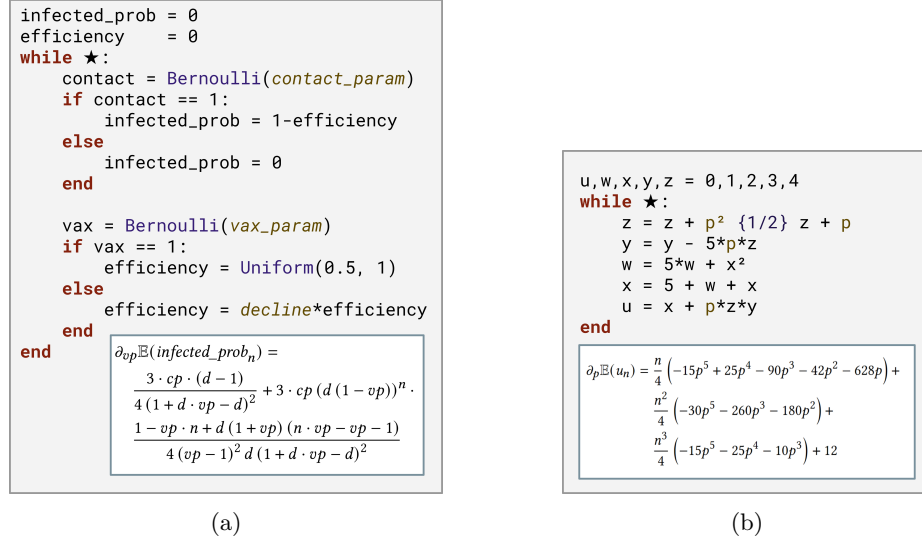


Fig. 1: Two examples of parameterized probabilistic loops, where our approach automatically derives loop sensitivities ∂_p as polynomial expressions depending on the loop counter n and other parameters; for example `infected_prob` with respect to `vax_param` (Fig. 1a) or that of `u` with respect to `p` (Fig. 1b). Using these results, our work shows that, when assuming `decline`=0.9, `contact_param`=0.7, after $n = 10$ time steps and currently having `vax_param`=0.1, then a small change ε in `vax_param` will decrease `infected_prob` by approximately 1.7ε in the next time step of Fig. 1a.

The sensitivity analysis of probabilistic programs is however hard due to their intrinsic randomness: program variables are no longer assigned single values but rather hold probability distributions [6]. Uncountably infinite state spaces and non-linear assignments are further obstacles to the formal analysis of probabilistic programs. In recent years, several frameworks to *manually* reason about the sensitivity of probabilistic programs were proposed [1,4,32]. However, the state-of-the-art in *automated* sensitivity analysis mainly focuses on loop-free programs such as *Bayesian networks* [13,14,9,31] and statically-bounded loops [21]. The technique presented in [33] supports loops with variable-dependent termination times, but can only verify that the sensitivities obey certain bounds. To the best of our knowledge, up to now, there is no automated and exact method supporting the sensitivity analysis of (potentially) unbounded probabilistic loops.

In this paper, we propose a fully automatic technique for the sensitivity analysis of unbounded probabilistic loops. The crux of our approach lies within the integration of methods from symbolic computation, probability theory and static analysis in order to automatically capture sensitivity information about probabilistic loops. Such an integrated framework allows us to also characterize a class of loops for which our technique is sound and complete.

Our framework for algebraic sensitivity analysis. We advocate the use of algebraic recurrences to model the behavior of probabilistic loops. We combine and adjust techniques from symbolic summation, partial derivatives, and probability theory to provide a step towards the exact and automated sensitivity analysis of probabilistic loops, even in the presence of uncountable state spaces and polynomial assignments. Figure 1 shows two probabilistic loops for which our work automatically computes the sensitivities of program variables with respect to different parameters. For example, Fig. 1a depicts a probabilistic program, modelling the incidence of a disease within a population. More precisely, it models the probability `infected_prob` that a single organism within the population is infected, in dependence on symbolic parameters that model the amount of social interaction (`contac_param`), the frequency of vaccinations (`vax_param`) and effect of a vaccination weakening over time (`decline`). Sensitivity analysis helps to reason about the influence of these parameters on the disease infection process, answering for example the question “How will an increase in the rate of vaccinations `vax_param` influence the probability `infected_prob` of an infection?”. Our work provides an algebraic approach to answering such and similar questions.

In a nutshell, our technique computes exact closed-form solutions for the sensitivities of (higher) moments of program variables for all, possibly infinitely many, loop iterations. Higher moments are necessary to recover/estimate the value distributions of probabilistic loop variables and hence these moments help in inferring valuable sensitivity information for the variance or skewness. In our work, we utilize algebraic techniques in probabilistic loop analysis to model moments of program variables with linear recurrences, so-called *moment recurrences* [26,8]. However, moment recurrences do not support loops with intricate polynomial arithmetic, such as the loop in Figure 1b. To overcome this limitation, we propose the notion of *sensitivity recurrences*, which shortcut computing closed-forms for variable moments and directly model sensitivities via linear recurrence equations. In Figure 1b, the program variable w is independent of the parameter p . By exploiting the independence of program variables from parameters, *sensitivity recurrences* enable the exact sensitivity analysis for loops such as Figure 1b. We characterize a class of probabilistic loops for which we prove *sensitivity analysis via sensitivity recurrences* to be sound and complete.

Our contributions. We integrate symbolic computation, in particular symbolic summation and partial derivation, in combination with methods from probability theory into the landscape of probabilistic program reasoning. In particular, we argue that recurrence-based loop analysis yields a fully automated and precise way to derive sensitivity information over unknown symbolic parameters in probabilistic loops. As such, our paper brings the following main contributions:

- We propose a fully automated approach for the sensitivity analysis of probabilistic loops based on *moment recurrences* (Section 3.1).
- We introduce *sensitivity recurrences* and an algorithm for sensitivity analysis going beyond *moment recurrences* (Section 3.2, Algorithm 1).

- We provide a precise characterization of the class of probabilistic loops for which *sensitivity recurrences* are provably sound and complete (Theorem 2).
- We describe an experimental evaluation demonstrating the feasibility of our techniques on many interesting probabilistic programs (Section 4).

2 Preliminaries

We write \mathbb{N} for the natural numbers, \mathbb{R} for the reals, $\overline{\mathbb{Q}}$ for the algebraic numbers, and $\mathbb{K}[x_1, \dots, x_k]$ for the polynomial ring with coefficients in the field \mathbb{K} . A polynomial consisting of a single monic term is a *monomial*. The expected value operator is denoted as \mathbb{E} .

2.1 Syntax and Semantics of Probabilistic Loops

Syntax. In this paper, we focus on unbounded probabilistic while-loops, as illustrated by the two examples of Figure 1 and introduced in [26]. Our programming model considers non-nested while-loops preceded by a variable initialization part, with the loop body being a sequence of (nested) if-statements and variable assignments. Unbounded probabilistic loops occur frequently when modeling dynamical systems. Guarded loops **while** G : **body** can be analyzed by considering the limiting behavior of unbounded loops of the form **while** **true**: **if** G : **body**.

The right-hand side of every variable assignment is either a probability distribution with existing moments (e.g. Normal or Uniform) and constant parameters, or a probabilistic choice of polynomials in program variables, that is $x = poly_1\{p_1\} \dots poly_k\{p_k\}$, where x is assigned to $poly_i$ with probability p_i . Further, programs can be parameterized by symbolic constants which represent arbitrary real numbers. For further details, we refer to Appendix A.

Throughout this paper, we refer to programs from our programming model simply by (probabilistic) loops or (probabilistic) programs. For a program \mathcal{P} we denote the set of program variables by $\text{Vars}(\mathcal{P})$ and the set of symbolic parameters by $\text{Params}(\mathcal{P})$.

Dependencies between program variables is a syntactical notion introduced next, representing a central part in our work.

Definition 1 (Variable Dependency). *Let \mathcal{P} be a probabilistic loop and $x, y \in \text{Vars}(\mathcal{P})$. We say that x depends directly on y , and write $x \rightarrow y$, if y appears in an assignment of x or an assignment of x occurs in an if-statement where y appears in the if-condition. Furthermore, we say that the dependency is non-linear, denoted as $x \xrightarrow{N} y$, if y appears non-linearly in an assignment of x .*

By \rightarrow we denote the transitive closure of \rightarrow . Regarding non-linearity, we write $x \xrightarrow{N} y$, if at least one of the direct dependencies from x to y is non-linear.

Example 1. In Figure 1b, we have (among others) $y \rightarrow z$, $w \xrightarrow{N} x$, $u \xrightarrow{N} w$, and $w \xrightarrow{N} u$. To illustrate the influence of if-conditions, in Figure 1a, note that $efficiency \rightarrow var$ and $infected_prob \rightarrow var$.

Semantics. Operationally, every probabilistic loop models an infinite-state Markov chain, which in turn induces a canonical probability space. Due to brevity, we omit the straightforward but rather technical construction of the Markov chains associated to probabilistic loops. For more details, we refer the interested reader to [26,16]. For an arithmetic expression $Expr$ in program variables, we denote by $Expr_n$ the stochastic process evaluating $Expr$ after the n th loop iteration.

2.2 C-finite Recurrences

We recall notions from algebraic recurrences [17,23], adjusted to our work.

A *sequence* of algebraic numbers is a function $u: \mathbb{N} \rightarrow \overline{\mathbb{Q}}$, succinctly denoted by $\langle u(n) \rangle_{n=0}^\infty$ or $\langle u(n) \rangle_n$. A *recurrence* for the sequence u of order $\ell \in \mathbb{N}$ is specified by a function $f: \mathbb{R}^{\ell+1} \rightarrow \mathbb{R}$ and given by the equation $u(n+\ell) = f(u(n+\ell-1), \dots, u(n+1), u(n), n)$. The *solutions* of a recurrence are the sequences satisfying the recurrence equation. Of particular relevance to our work is the class of *linear recurrences with constant coefficients* or more shortly, *C-finite recurrences*. The sequence u satisfies a C-finite recurrence if $u(n+\ell) = c_{\ell-1}u(n+\ell-1) + c_{\ell-2}u(n+\ell-2) + \dots + c_0u(n)$ holds, where $c_0, \dots, c_{\ell-1} \in \overline{\mathbb{Q}}$ are constants and $c_0 \neq 0$. Every C-finite recurrence is associated with its *characteristic polynomial* $x^\ell - c_{\ell-1}x^{\ell-1} - \dots - c_1x - c_0$. The solutions of C-finite recurrences can always be computed [23] and written in closed-form as *exponential polynomials*. More precisely, if $\langle u(n) \rangle_n$ is the solution to a C-finite recurrence, then $u(n) = \sum_{k=1}^r P_k(n)\lambda_k^n$ where $P_k(n) \in \overline{\mathbb{Q}}[n]$ and $\lambda_1, \dots, \lambda_r$ are the roots of the characteristic polynomial. The properties of C-finite recurrences also hold for systems of C-finite recurrences (systems of linear recurrence equations with constant coefficients, specifying multiple sequences).

2.3 Higher Moment Analysis using Recurrences

For a random variable x , its higher moments are defined as $\mathbb{E}(x^k)$ for $k \in \mathbb{N}$. More generally, mixed moments for a set of random variables S are expected values of monomials in S . Recent works in probabilistic program analysis [26,10] introduced techniques and tools based on C-finite recurrences to compute higher moments of program variables for probabilistic loops. For example, for a probabilistic loop, $k \in \mathbb{N}$ and a program variable x , a closed-form solution for the k th higher moment of x parameterized by the loop iteration n , that is $\mathbb{E}(x_n^k)$, is computed in [26] using the **Polar** tool. This is achieved by first normalizing the program to eliminate if-statements and ensure every variable is only assigned once in the loop body. Then, a system of C-finite recurrences is constructed that models expected values of monomials in program variables. More precisely, for a monomial M in program variables, the work of [26] constructs a linear recurrence equation, relating the expected value of M in iteration $n+1$ to the expected values of program variable monomials in iteration n . The linear recurrence for the expected value of M in iteration $n+1$ is constructed by starting with the expression $\mathbb{E}(M_{n+1})$ and replacing variables contained in the expression by their assignments bottom-up as they appear in the loop body. Throughout,

the linearity of expectation is used to convert expected values of polynomials into expected values of monomials (cf. Appendix C).

We adopt the setting of [26,10] and refer by *moment recurrences* to the recurrence equations these techniques construct for moments of program variables.

Definition 2 (Moment Recurrence). *Let \mathcal{P} be a probabilistic loop and M a monomial in $\text{Vars}(\mathcal{P})$. A moment recurrence for M is an equation $\mathbb{E}(M_{n+1}) = \sum_{i=1}^r c_i \cdot \mathbb{E}(W_n^{(i)})$ where $c_i \in \overline{\mathbb{Q}}$ and all $W^{(i)}$ are monomials in $\text{Vars}(\mathcal{P})$.*

In order to compute a closed-form solution for $\mathbb{E}(x_n^k)$, we employ [26] to first compute a moment recurrence R for the monomial x^k . Next, we derive moment recurrences for all monomials $W^{(i)}$ in R (cf. Definition 2) to construct a system of C-finite recurrences.

Example 2. Consider the program from Figure 1a. For a more succinct representation, we abbreviate the symbolic parameters as $cp := \text{contact_param}$; $vp := \text{vax_param}$ and $d := \text{decline}$. The first moments of the program variables are modeled through the following system of C-finite recurrences [26]:

$$\begin{aligned}\mathbb{E}(\text{infected_prob}_{n+1}) &= cp - cp \cdot \mathbb{E}(\text{efficiency}_n) \\ \mathbb{E}(\text{efficiency}_{n+1}) &= (d - d \cdot vp) \cdot \mathbb{E}(\text{efficiency}_n) + \frac{3}{4} \cdot vp\end{aligned}$$

The initial values of $\mathbb{E}(\text{infected_prob}_n)$ and $\mathbb{E}(\text{efficiency}_n)$ are both 0. The system can be automatically solved [23] to obtain closed-form solutions, which are, when expanded, exponential polynomials, e.g. for $\mathbb{E}(\text{infected_prob}_n)$:

$$\mathbb{E}(\text{infected_prob}_n) = cp + \frac{3 \cdot vp \cdot cp \cdot ((d - d \cdot vp)^{n-1} - 1)}{4(d \cdot vp - d + 1)}$$

We note that moment recurrences do not always exist. Moreover, termination is not guaranteed when recursively inferring the moment recurrences for all monomials $W^{(i)}$ in Definition 2 in order to construct a C-finite system.

Example 3. To illustrate that the approach based on moment recurrences does not work unconditionally, consider the loop from Figure 1b and construct the moment recurrence $\mathbb{E}(w_{n+1}) = 5 \cdot \mathbb{E}(w_n) + \mathbb{E}(x_n^2)$. Since the recurrence contains $\mathbb{E}(x_n^2)$, we require the moment recurrence $\mathbb{E}(x_{n+1}^2) = \mathbb{E}((5 + w_{n+1} + x_n)^2) = \mathbb{E}(w_{n+1}^2) + \dots$ which requires the recurrence for $\mathbb{E}(w_n^2)$. This in turn necessitates a recurrence for $\mathbb{E}(x_n^4)$, which necessitates the recurrence for $\mathbb{E}(w_n^4)$ and so on. This process will repeatedly require recurrences for increasing moments of x_n and w_n , implying that this process will not terminate.

To circumvent variable dependencies and compute closed-forms of moment recurrences, we note that the following two conditions on the probabilistic loops ensure existence and computability of higher order moments.

Definition 3 (Admissible Loop). *A loop is admissible if*

1. all variables in branching conditions only assume values in a finite set (i.e. they are finite valued), and
2. no variable x is non-linearly self-dependent ($x \not\stackrel{N}{\rightarrow} x$)¹.

Example 4. The probabilistic loop in Figure 1a is admissible. However, the program in Figure 1b is not admissible. It does not satisfy condition 2: the variable x depends linearly on w and w depends quadratically on x ; therefore, x is non-linearly self-dependent.

Admissible probabilistic loops are *moment-computable* [26], that is, higher moments of program variables admit computable closed-forms as exponential polynomials. The restriction on finite valued variables in branching conditions is necessary to guarantee computability and completeness: a single branching statement involving an unbounded variable renders the program model Turing-complete [26].

3 Sensitivity Analysis

In this section, we study the sensitivity of program variable moments with respect to symbolic parameters. We present two exact and fully automatic methods to answer the question of how small changes in symbolic parameters influence the moments of program variables. As such, we exploit the fact that closed-forms for variable moments in admissible loops are computable (Section 3.1). We further go beyond the admissible loop setting (Section 3.2) and devise a sensitivity analysis technique applicable to some non-admissible loops, such as the program in Figure 1b.

Definition 4 (Sensitivity). Let \mathcal{P} be a probabilistic loop, $x \in \text{Vars}(\mathcal{P})$ and $p \in \text{Params}(\mathcal{P})$. The sensitivity of the k th moment of x with respect to p , denoted as $\partial_p \mathbb{E}(x_n^k)$, is defined as the partial derivative of $\mathbb{E}(x_n^k)$ with respect to p , and parameterized by loop counter n . For monomials M of variables, the sensitivity $\partial_p \mathbb{E}(M_n)$ is defined analogously.

Similar to *moment computability* [26], we define a program to be *sensitivity computable* if the sensitivities of all the variables' expected values are expressible in closed-form.

Definition 5 (Sensitivity Computability). Let \mathcal{P} be a probabilistic program and $p \in \text{Params}(\mathcal{P})$. \mathcal{P} is sensitivity computable with respect to p , if for every variable $x \in \text{Vars}(\mathcal{P})$ the sensitivity $\partial_p \mathbb{E}(x_n)$ has an exponential polynomial closed-form that is computable.

¹ While [26] allows arbitrary dependencies among finite valued variables, our work omits this generalization for simplicity. Nevertheless, our results also apply to admissible loops with arbitrary dependencies among finite valued variables.

3.1 Sensitivity Analysis for Admissible Loops

As mentioned in Section 2, for admissible loops, any moment of every program variable admits a closed-form solution as an exponential polynomial which is computable. That is, for a program variable x and $k \in \mathbb{N}$, the k th moment of x can be written as $\mathbb{E}(x_n^k) = \sum_{j=0}^r P_j(n) \lambda_j^n$, where $P_j \in \overline{\mathbb{Q}}[n]$ and $\lambda_j \in \overline{\mathbb{Q}}$ may contain symbolic parameters. We next show that based on the closed-forms of variable moments, we can compute exponential polynomials representing the sensitivities of moments on parameters.

Theorem 1 (Admissible Sensitivities). *Let \mathcal{P} be an admissible program, $x \in \text{Vars}(\mathcal{P})$, $p \in \text{Params}(\mathcal{P})$, and $k \in \mathbb{N}$. Then, the sensitivity $\partial_p \mathbb{E}(x_n^k)$ has an exponential polynomial closed-form that is computable.*

Proof. Because \mathcal{P} is admissible, $\mathbb{E}(x_n^k)$ can be expressed as an exponential polynomial. We show that the sensitivity can be expressed as an exponential polynomial by expanding $\mathbb{E}(x_n^k)$ into a sum of exponential monomials: $\mathbb{E}(x_n^k) = \sum_{j=0}^r P_j(n) \lambda_j^n = \sum_{j=0}^r \sum_{i=0}^{m_j} M_{ij}(n) \lambda_j^n$, where m_j is the number of monomials in P_j and every M_{ij} is a monomial. Note that every M_{ij} and λ_j may depend on the symbolic constant p . The derivative of the exponential monomials can then be obtained by applying the product rule for derivatives:

$$\begin{aligned} \partial_p \mathbb{E}(x_n^k) &= \sum_{j=0}^r \sum_{i=0}^{m_j} (\partial_p M_{ij}(n)) \lambda_j^n + M_{ij}(n) \cdot n \cdot (\partial_p \lambda_j) \cdot \lambda_j^{n-1} \\ &= \sum_{j=0}^r (\partial_p P_j(n) + P_j(n) \cdot n \cdot \partial_p \lambda_j \cdot \frac{1}{\lambda_j}) \lambda_j^n \end{aligned}$$

It is left to show that the exponential polynomial $\partial_p \mathbb{E}(x_n^k)$ is computable. Because \mathcal{P} is admissible, an exponential polynomial for $\mathbb{E}(x_n^k)$ is computable. Now, the second claim follows from the fact that exponential polynomials are elementary and that the derivative of any elementary function is computable. \square

As a corollary, admissible loops are sensitivity computable. Although *sensitivity computability* only refers to first moments, Theorem 1 shows that for admissible loops, sensitivities of *all* higher moments of program variables admit a computable closed-form.

Example 5. Consider Figure 1a. In Example 2 we stated the closed-form solutions of $\mathbb{E}(\text{infected_prob}_n)$. The sensitivities of the respective expected values can be computed by symbolic differentiation and, by Theorem 1, can be expanded to exponential polynomials. For example, the following expression describes the sensitivity of $\mathbb{E}(\text{infected_prob}_n)$ with respect to the parameter vp :

$$\begin{aligned} \partial_{vp} \mathbb{E}(\text{infected_prob}_n) &= \frac{3 \cdot cp (1 - vp \cdot n + d (1 + vp) (n \cdot vp - vp - 1)) (d (1 - vp))^n}{4 (vp - 1)^2 d (1 + d \cdot vp - d)^2} \\ &\quad + \frac{3 \cdot cp \cdot (d - 1)}{4 (1 + d \cdot vp - d)^2} \end{aligned}$$

3.2 Sensitivity Analysis for Non-Admissible Loops

In general, moments of program variables of non-admissible loops do not satisfy linear recurrences. Therefore, we cannot utilize closed-forms of the moments for sensitivity analysis. Nevertheless, sensitivity analysis is feasible even for some non-admissible loops. In this section, we propose a novel sensitivity analysis approach applicable to non-admissible loops. Moreover, we characterize the class of (non-admissible) loops for which our method is sound and complete.

For admissible loops, linear recurrences describing variable moments can be used as an intermediary step to compute sensitivities. The core of our approach towards handling non-admissible loops is to shortcut moment recurrences and devise recurrences directly for sensitivities. Due to independence with respect to the sensitivity parameter, sensitivities of program variables can follow a linear recurrence even though their moments do not. We illustrate the idea of our new method on the non-admissible loop from Figure 1b.

Example 6. Consider the non-admissible program from Figure 1b. The moment recurrences for all program variables are:

$$\begin{aligned}\mathbb{E}(z_{n+1}) &= \mathbb{E}(z_n) + 0.5 \cdot (p + p^2) & \mathbb{E}(y_{n+1}) &= \mathbb{E}(y_n) - 5p \cdot \mathbb{E}(z_{n+1}) \\ \mathbb{E}(w_{n+1}) &= 5 \cdot \mathbb{E}(w_n) + \mathbb{E}(x_n^2) & \mathbb{E}(x_{n+1}) &= 5 + \mathbb{E}(w_{n+1}) + \mathbb{E}(x_n) \\ \mathbb{E}(u_{n+1}) &= \mathbb{E}(x_{n+1}) + p \cdot \mathbb{E}(zy_{n+1})\end{aligned}$$

As illustrated in Example 3, we cannot complete the recurrences to a C-finite system because both w and x are non-linearly self-dependent. Therefore, we cannot compute closed-form solutions for $\mathbb{E}(w_n)$ and $\mathbb{E}(x_n)$. However, we can shortcut solving for $\mathbb{E}(w_n)$ and $\mathbb{E}(x_n)$ by differentiating the moment recurrences with respect to p and establish recurrences directly for the sensitivities:

$$\begin{aligned}\partial_p \mathbb{E}(z_{n+1}) &= \partial_p \mathbb{E}(z_n) + 0.5 \cdot (1 + 2p) \\ \partial_p \mathbb{E}(y_{n+1}) &= \partial_p \mathbb{E}(y_n) - 5p \cdot \partial_p \mathbb{E}(z_{n+1}) - 5 \cdot \mathbb{E}(z_{n+1}) \\ \partial_p \mathbb{E}(w_{n+1}) &= 5 \cdot \partial_p \mathbb{E}(w_n) + \partial_p \mathbb{E}(x_n^2) \\ \partial_p \mathbb{E}(x_{n+1}) &= \partial_p \mathbb{E}(w_{n+1}) + \partial_p \mathbb{E}(x_n) \\ \partial_p \mathbb{E}(u_{n+1}) &= \partial_p \mathbb{E}(x_{n+1}) + \mathbb{E}(zy_{n+1}) + p \cdot \partial_p \mathbb{E}(zy_{n+1})\end{aligned}$$

Now, because the variables w and x do not depend on the parameter p , we conclude that $\partial_p \mathbb{E}(w_n) \equiv \partial_p \mathbb{E}(x_n) \equiv 0$. The sensitivity recurrences thus simplify:

$$\begin{aligned}\partial_p \mathbb{E}(z_{n+1}) &= \partial_p \mathbb{E}(z_n) + \frac{1 + 2p}{2} \\ \partial_p \mathbb{E}(y_{n+1}) &= \partial_p \mathbb{E}(y_n) - 5p \cdot \partial_p \mathbb{E}(z_{n+1}) - 5 \cdot \mathbb{E}(z_{n+1}) \\ \partial_p \mathbb{E}(u_{n+1}) &= \mathbb{E}(zy_{n+1}) + p \cdot \partial_p \mathbb{E}(zy_{n+1})\end{aligned}$$

We can interpret sensitivities such as $\partial_p \mathbb{E}(z_n)$ or $\partial_p \mathbb{E}(u_n)$ as atomic recurrence variables. In the resulting recurrences, all variables with non-linear self-dependencies vanished. Therefore, the recurrences can be completed to a C-finite

system and solved by existing techniques, even though $\mathbb{E}(w_n)$ and $\mathbb{E}(x_n)$ are not C-finite. The resulting system of recurrences consists of all recurrences for sensitivities and moments that appear on the right-hand side of another recurrence. That is, the system of recurrences consists of the sensitivity recurrences for $\partial_p \mathbb{E}(z)$, $\partial_p \mathbb{E}(y)$, $\partial_p \mathbb{E}(u)$, $\partial_p \mathbb{E}(yz)$, $\partial_p \mathbb{E}(z^2)$ and the moment recurrences for $\mathbb{E}(z)$, $\mathbb{E}(y)$, $\mathbb{E}(yz)$, $\mathbb{E}(z^2)$.

Motivated by Example 6, we introduce the notion of *sensitivity recurrences*.

Definition 6 (Sensitivity Recurrence). *Let \mathcal{P} be a program, $p \in \text{Params}(\mathcal{P})$ a symbolic parameter, M a monomial in $\text{Vars}(\mathcal{P})$ and let $\mathbb{E}(M_{n+1}) = \sum_{i=1}^r c_i \cdot \mathbb{E}(W_n^{(i)})$ be the moment-recurrence of M . Then the sensitivity recurrence of M with respect to p is defined as*

$$\boxed{\partial_p \mathbb{E}(M_{n+1})} := \frac{\partial \mathbb{E}(M_{n+1})}{\partial p} = \frac{\partial}{\partial p} \left(\sum_{i=1}^r c_i \cdot \mathbb{E}(W_n^{(i)}) \right) \quad (1)$$

$$\boxed{= \sum_{i=1}^r \left(\frac{\partial}{\partial p} c_i \right) \cdot \mathbb{E}(W_n^{(i)}) + c_i \cdot \partial_p \mathbb{E}(W_n^{(i)})}$$

The sensitivity recurrence of M equates the sensitivity of M at iteration $n+1$ to moments *and sensitivities* at iteration n . Along the ideas in Example 6, we provide with Algorithm 1 a procedure for sensitivity analysis also applicable to non-admissible loops. The idea of Algorithm 1 is to determine $\partial_p \mathbb{E}(M_n)$ by constructing a C-finite system consisting of all necessary recurrence equations for the moments and sensitivities of program variables. As illustrated in Example 6, we can exploit the independence of variables from the sensitivity parameter p to simplify the problem: if a monomial W' is independent from p then $\partial_p \mathbb{E}(W'_n) \equiv 0$. Moreover, if p does not appear in the constant c_i of Equation (1), then $(\partial/\partial p)c_i = 0$, and hence the moment recurrence of W' does not need to be constructed (lines 8–9 of Algorithm 1). This is essential if the *expected value* of W' does not admit a closed-form. Algorithm 1 is sound by construction, however, termination is non-trivial. In the remainder of this section, we formalize the notion of parameter (in)dependence and give a characterization of the class of non-admissible loops for which Algorithm 1 terminates. As a consequence of Algorithm 1, we show that sensitivity recurrences yield an exact and complete technique for sensitivity analysis (Theorem 2).

Definition 7 (p -Dependent Variable). *Let \mathcal{P} be a program with parameter $p \in \text{Params}(\mathcal{P})$. A variable $x \in \text{Vars}(\mathcal{P})$ is p -dependent, if (1) p appears in an assignment of x , (2) x depends on some $y \in \text{Vars}(\mathcal{P})$ ($x \rightarrow y$) and y is p -dependent or (3) an assignment of x occurs in an if-statement where p appears in the if-condition. A variable is p -independent if it is not p -dependent. A monomial M in program variables is p -dependent if M contains at least one p -dependent variable, otherwise it is p -independent.*

Algorithm 1 Computing Sensitivities via Sensitivity Recurrences

Input: program \mathcal{P} , monomial M in $\text{Vars}(\mathcal{P})$, $p \in \text{Params}(\mathcal{P})$ **Output:** closed-form for $\partial_p \mathbb{E}(M_n)$

```

1: if  $M$  is  $p$ -independent then
2:   return 0
3: end if
4:  $Eqs \leftarrow \emptyset$ ;  $Mom \leftarrow \emptyset$ ;  $Sens \leftarrow \{M\}$ 
5: while  $Sens \neq \emptyset$  do ▷ Add all necessary sensitivity recurrences
6:   pick  $W \in Sens$ ;  $Sens \leftarrow Sens \setminus \{W\}$ 
7:    $SRec \leftarrow$  sensitivity recurrence of  $W$ 
8:   Replace every  $\partial_p \mathbb{E}(W'_n)$  in  $SRec$  by 0 if  $W'$  is  $p$ -independent
9:   Replace every  $(\partial/\partial p c) \mathbb{E}(W'_n)$  in  $SRec$  by 0 if  $(\partial/\partial p c) = 0$ 
10:   $Eqs \leftarrow Eqs \cup \{SRec\}$ 
11:  Add to  $Sens$  all monomials  $W'$  s.t.  $\partial_p \mathbb{E}(W'_n)$  in  $SRec$ 
12:   $\hookrightarrow$  and the sensitivity recurrence of  $W' \notin Eqs$ 
13:  Add to  $Mom$  all monomials  $W'$  s.t.  $\mathbb{E}(W'_n)$  in  $SRec$ 
14: end while
15: while  $Mom \neq \emptyset$  do ▷ Add all necessary moment recurrences
16:   pick  $W \in Mom$ ;  $Mom \leftarrow Mom \setminus \{W\}$ 
17:    $MRec \leftarrow$  moment recurrence of  $W$ 
18:    $Eqs \leftarrow Eqs \cup \{MRec\}$ 
19:   Add to  $Mom$  all monomials  $W'$  s.t.  $\mathbb{E}(W'_n)$  in  $MRec$ 
20:    $\hookrightarrow$  and the moment recurrence of  $W' \notin Eqs$ 
21: end while
22:  $S \leftarrow$  solve system of C-finite recurrences  $Eqs$ 
23: return closed-form of  $\partial_p \mathbb{E}(M_n)$  from  $S$ 

```

For any p -independent monomial M in program variables, the corresponding sensitivity $\partial_p \mathbb{E}(M_n)$ is zero (by using induction on n and applying Definition 7).

Lemma 1. *Let \mathcal{P} be a program, $p \in \text{Params}(\mathcal{P})$ a symbolic parameter and M a p -independent monomial in $\text{Vars}(\mathcal{P})$, then it holds that the sensitivity variable of M is zero, i.e., $\forall n \geq 0 : \partial_p \mathbb{E}(M_n) = 0$.*

In Example 6, the moments $\mathbb{E}(w_n)$ and $\mathbb{E}(x_n)$ do not admit closed-forms. We resolved this issue by differentiating all moment recurrences and working directly with the sensitivity recurrences, where the moment recurrences for w and x vanished. Crucial for this phenomenon is the fact that the variables w and x are independent of the sensitivity parameter p .

However, a second fact is necessary to guarantee that the moment recurrences of w and x do not appear in the resulting system of recurrences: Assume some new variable v depends on x and has the moment recurrence $\mathbb{E}(v_{n+1}) = \mathbb{E}(v_n) + p \cdot \mathbb{E}(x_n)$. Then the sensitivity recurrence for v is given by $\partial_p \mathbb{E}(v_{n+1}) = \partial_p \mathbb{E}(v_n) + \mathbb{E}(x_n) + p \cdot \partial_p \mathbb{E}(x_n)$. Even though x itself is p -independent, $\mathbb{E}(x_n)$ remains in the sensitivity recurrence of v because the coefficient of $\mathbb{E}(x_n)$ contains the parameter p . A similar effect occurs if the moment recurrence for v was $\mathbb{E}(v_{n+1}) = \mathbb{E}(v_n) + \mathbb{E}(z_n x_n)$, because z is p -dependent.

Our goal is to characterize the class of probabilistic loops for which sensitivity recurrences yield a sound and complete method for sensitivity analysis. Hence, we need to capture the notion that some dependencies between variables are free of multiplicative factors involving the sensitivity parameter. We do this in the following definition by refining our dependency relation \rightarrow .

Definition 8 (p -Influenced Dependency). *Let \mathcal{P} be a program with parameter $p \in \text{Params}(\mathcal{P})$ and $x, y \in \text{Vars}(\mathcal{P})$ with $x \rightarrow y$. Then, the direct dependency between x and y is p -influenced, written as $x \rightarrow_p y$, if at least one of the following conditions hold:*

- *An assignment of x contains y and occurs in an if-statement with the if-condition involving p or a p -dependent variable.*
- *An assignment of x contains y and is a probabilistic choice with some probability of the choice depending on p .*
- *An assignment of x contains a term $c \cdot M \cdot y$ where c is constant and M is a monomial in program variables (possibly containing y). Moreover, either c contains p or M contains a p -dependent variable.*

If $x \rightarrow y$, we write $x \rightarrow_p y$ if some dependency from x to y is p -influenced. If $x \rightarrow y$ and $x \not\rightarrow_p y$ we call the dependency between x and y p -free.

Definition 8 covers all cases in the construction of moment recurrences that introduce multiplicative factors depending on the sensitivity parameter p [26]. We provide details on the construction of moment recurrences in Appendix C.

More concretely, assume \mathcal{P} to be a program and $x \in \text{Vars}(\mathcal{P})$. The moment recurrence of x contains expected values of monomials M of program variables.

Additionally, the moment recurrences of any M will again contain expected values of monomials of program variables and so on. We capture all of these monomials with the notion of *descendant monomials* in Definition 9. Intuitively, to construct a system of moment recurrences for $\mathbb{E}(x_n)$ one needs to include the moment recurrences of all descendants of x .

Definition 9 (Descendant Monomial). *Let \mathcal{P} be a program, $x \in \text{Vars}(\mathcal{P})$, and M a monomial in program variables. The monomial M is a descendant of the variable x if (1) $M = x$, or (2) M occurs in the moment recurrence of a monomial W and W is a descendant of x . The variable x is an ancestor of M .*

There is a dependency between x and any variable of any descendant of x , which means $x \rightarrow y$ for every descendant M of x and every variable y in M . Our dependency relation from Definition 8 allows us to pinpoint the variables in the moment recurrence of any descendant of x (Definition 9) with a multiplicative factor involving the sensitivity parameter. Definitions 8 and 9 together with the procedure constructing moment recurrences (Appendix C) yield:

Lemma 2 (p -Influenced Moment Recurrence). *Let \mathcal{P} be a program, $x \in \text{Vars}(\mathcal{P})$, and $p \in \text{Params}(\mathcal{P})$. Assume M is a monomial in program variables descending from x . Let W be a monomial in M 's moment recurrence with non-zero coefficient c . If the parameter p occurs in c , then for all variables y in W we have $x \rightarrow_p y$. Moreover, if some variable z in W is p -dependent, then for all variables y in W different from z we have $x \rightarrow_p y$.*

We now state our main result (Theorem 2) describing the class of probabilistic loops for which Algorithm 1 terminates and, hence, sensitivity recurrences are sound and complete. We characterize the class of loops in terms of our dependency relations as well as variables with non-linear self-dependencies, which we refer to as *defective* variables.

Definition 10 (Defective Variables). *Let \mathcal{P} be a program and $x \in \text{Vars}(\mathcal{P})$, then x is defective if $x \xrightarrow{N} x$. Otherwise, x is effective.*

Theorem 2 (Non-Admissible Sensitivities). *Let \mathcal{P} be a probabilistic program, $p \in \text{Params}(\mathcal{P})$, $x \in \text{Vars}(\mathcal{P})$, and assume all the following conditions:*

1. *All variables occurring in branching conditions are finite.*
2. *All defective variables are p -independent.*
3. *All dependencies on defective variables are p -free.*

Then, for every monomial M in program variables descending from x , Algorithm 1 terminates on input \mathcal{P} , M and p .

Proof (Sketch). Algorithm 1 does not terminate iff infinitely many monomials are added to the set *Sens* one line 11 or to the set *Mom* on lines 13 or 19. However, every monomial added to these sets decreases with respect to some

well-founded ordering. Hence, only finitely many monomials are added and Algorithm 1 terminates. This holds by using a well-founded ordering for monomials of effective variables and showing that all monomials added to *Sens* or *Mom* do *not* contain defective variables. Assuming then that some monomial added to the sets *Sens* or *Mom* contains defective variables leads to contradictions using conditions 2 and 3, and Lemma 2. See Appendix D for more details. \square

Theorem 2 characterizes the class of probabilistic loops for which sensitivity recurrences provide a sound and complete method for sensitivity analysis. As an immediate corollary, this class of loops is sensitivity computable because every variable is a descendant of itself. Note that all conditions of Theorem 2 are statically checkable: the concepts of defective variables, p -independent variables, and p -free dependencies are purely syntactic notions. Moreover, program variables occurring in branching conditions only admitting finitely many values can be verified using standard techniques based on *abstract interpretation*.

Theorem 2 also applies to sensitivity analysis for higher moments: let $v \in \text{Vars}(\mathcal{P})$ and $k \in \mathbb{N}$, then Theorem 2 covers the sensitivity of v ’s k th moment if v^k is a descendant of some variable. Otherwise, v^k can be dealt with by introducing a fresh variable w and appending the assignment $w := v^k$ to \mathcal{P} ’s loop body.

The proof of Theorem 2 provides an alternative argument for admissible loops being sensitivity computable (Theorem 1); as admissible loops do not contain defective variables by definition (Definition 3), the class of loops characterized by Theorem 2 subsumes the class of admissible loops.

4 Experiments and Evaluation

We evaluate our methods for sensitivity analysis for admissible loops (Section 3.1) and non-admissible loops (Section 3.2). Our techniques for sensitivity analysis extend the **Polar** framework [26], which is publicly available at <https://github.com/probing-lab/polar>. For admissible loops, we use the existing functionality of the **Polar** framework to compute closed-forms for the moments of program variables.

Experimental Setup. We split our evaluation into two parts. First, we compute the sensitivities of (higher) moments of program variables for admissible loops by automatically differentiating the closed-forms of the variables’ moments (Table 1). In the second part, we consider our method using sensitivity recurrences, which is also applicable to non-admissible loops (Table 2). To the best of our knowledge, our work provides the first exact and fully automatic tool to compute the sensitivities of (higher) moments of program variables for probabilistic loops. All our experiments have been executed on a machine with a 2.6 GHz Intel i7 (Gen 10) processor and 32 GB of RAM with a timeout (TO) of 120 s.

Differentiating Closed-Forms. Table 1 shows the evaluation of our sensitivity analysis technique for admissible loops (Section 3) on 11 benchmarks. The benchmarks consist of the running example from Figure 1a and parameterized probabilistic loops from the benchmarks in [26], coming from literature on probabilistic

program analysis [3,12,20,9]. All the benchmarks contain at least one symbolic parameter with respect to which the sensitivities are computed. Table 1 shows that our approach is capable of computing the sensitivities of higher moments of program variables for challenging loops with various characteristics, such as discrete and continuous state spaces as well as drawing from common distributions.

BENCHMARK	SENSITIVITY	REC, RT	SENSITIVITY	REC, RT
50-Coin-Flips	$\partial_p \mathbb{E}(\text{total})$	51, 1.56	$\partial_p \mathbb{E}(\text{total}^2)$	TO, TO
Bimodal	$\partial_p \mathbb{E}(x)$	3, 0.40	$\partial_p \mathbb{E}(x^2)$	5, 0.72
Component-Health	$\partial_{p1} \mathbb{E}(\text{obs}),$	2, 0.61	$\partial_{p1} \mathbb{E}(\text{obs}^2),$	2, 0.62
Umbrella	$\partial_{u1} \mathbb{E}(\text{umbrella})$	2, 0.97	$\partial_{u1} \mathbb{E}(\text{umbrella}^2)$	2, 0.98
Gambler’s Ruin	$\partial_p \mathbb{E}(\text{money})$	4, 11.2	$\partial_p \mathbb{E}(\text{money}^2)$	10, 64.6
Hawk-Dove	$\partial_v \mathbb{E}(\text{p1bal})$	1, 0.34	$\partial_v \mathbb{E}(\text{p1bal}^2)$	2, 0.67
Las-Vegas-Search	$\partial_p \mathbb{E}(\text{attempts})$	2, 0.57	$\partial_p \mathbb{E}(\text{attempts}^2)$	4, 7.31
1D-Random-Walk	$\partial_p \mathbb{E}(x)$	1, 0.27	$\partial_p \mathbb{E}(x^2)$	2, 0.39
2D-Random-Walk	$\partial_{p_right} \mathbb{E}(x)$	1, 0.28	$\partial_{p_right} \mathbb{E}(x^2)$	2, 0.41
Randomized-Response	$\partial_p \mathbb{E}(p1)$	1, 0.29	$\partial_p \mathbb{E}(p1^2)$	2, 0.42
Vaccination (Fig. 1a)	$\partial_{vp} \mathbb{E}(\text{infected})$	2, 1.25	$\partial_{vp} \mathbb{E}(\text{infected}^2)$	2, 1.19

Table 1: Evaluation of the sensitivity computation for 11 admissible loops by differentiating closed-forms of variable moments. REC: size of the recurrence system to compute the variables’ moments; RT: runtime in seconds; TO: timeout.

Sensitivity Recurrences. Table 2 shows the evaluation of our sensitivity analysis technique from Algorithm 1 using *sensitivity recurrences*. The benchmarks consist of four non-admissible loops and six admissible loops from Table 1. Non-admissible loops are known to be notoriously hard to analyze automatically [2]. Table 2 shows that *sensitivity recurrences* are capable of computing the sensitivities for admissible as well as non-admissible loops.

Experimental Summary. When comparing both approaches on admissible loops, the differentiation-based approach typically performs better, e.g., on the benchmarks “Gambler’s Ruin” or “Vaccination”. This is not surprising, as the main complexity in both approaches lies in solving the system of recurrences and when using sensitivity recurrences, the number of recurrences tends to be higher. However, the exact number of recurrences depends on the program structure, and as such, there are cases where the approach using sensitivity recurrences performs equally well, such as in the “Randomized-Response” benchmark. Nevertheless, for the class of loops characterized in Section 3.2, the differentiation-based approach fails, whereas sensitivity recurrences still deliver exact results in a fully automated manner.

BENCHMARK	SENSITIVITY	REC, RT	SENSITIVITY	REC, RT
Non-Admissible (Fig. 1b)	$\partial_p \mathbb{E}(u)$	9, 1.40	$\partial_p \mathbb{E}(y^2)$	9, 1.75
Non-Admissible-2	$\partial_{\text{par}} \mathbb{E}(y)$	5, 6.56	$\partial_{\text{par}} \mathbb{E}(xz)$	4, 3.67
Non-Admissible-3	$\partial_p \mathbb{E}(\text{total})$	6, 12.6	$\partial_p \mathbb{E}(z1^2)$	12, 56.5
Non-Admissible-4	$\partial_{p1} \mathbb{E}(z)$	4, 0.48	$\partial_{p1} \mathbb{E}(\text{cnt}^2)$	3, 0.39
Bimodal	$\partial_{\text{var}} \mathbb{E}(x)$	3, 0.28	$\partial_{\text{var}} \mathbb{E}(x^2)$	5, 0.42
Component-Health	$\partial_{p1} \mathbb{E}(\text{obs})$	3, 0.74	$\partial_{p1} \mathbb{E}(\text{obs}^2)$	3, 0.73
Gambler's Ruin	$\partial_p \mathbb{E}(\text{money})$	7, 66.9	$\partial_p \mathbb{E}(\text{money}^2)$	TO, TO
Las-Vegas-Search	$\partial_p \mathbb{E}(\text{attempts})$	3, 0.81	$\partial_p \mathbb{E}(\text{attempts}^2)$	7, 13.3
Randomized-Response	$\partial_p \mathbb{E}(p1)$	1, 0.30	$\partial_p \mathbb{E}(p1^2)$	3, 0.40
Vaccination (Fig. 1a)	$\partial_{vp} \mathbb{E}(\text{infected})$	3, 8.26	$\partial_{vp} \mathbb{E}(\text{infected}^2)$	3, 7.85

Table 2: Evaluation of the sensitivity computation for 10 loops (4 are non-admissible) using sensitivity recurrences. REC: size of the recurrence system to compute the variables' sensitivities; RT: runtime in seconds; TO: timeout.

Our experiments demonstrate that our novel techniques for sensitivity analysis can compute the sensitivities for a rich class of probabilistic loops with discrete and continuous state spaces, drawing from probability distributions, and including polynomial arithmetic. Moreover, the technique based on our new notion of *sensitivity recurrences* can compute sensitivities for probabilistic loops for which closed-forms of the variables' moments do not exist.

5 Related Work

Sensitivity & Probabilistic Programs. Bayesian networks can be seen as special loop-free probabilistic programs. The sensitivity of Bayesian networks with discrete probability distribution was studied in [13,14]. The works of [9,31] provide a framework to analyze properties (sensitivity among others) of *Prob-solvable Bayesian networks*. In contrast, our work focuses on probabilistic loops with more complex control flow and supports continuous distributions. In recent years, techniques emerged to manually reason about sensitivities of probabilistic programs, such as program calculi [1], custom logics [4], or type systems [32]. Although applicable to general probabilistic programs, these techniques require manual reasoning or user guidance, while our work focuses on full automation.

A fully-automatic and exact sensitivity analyzer for probabilistic programs with statically bounded loops was proposed in [21]. In comparison, our work focuses on potentially unbounded loops. The authors of [33] introduce an automatable approach for expected sensitivity based on martingales. Their technique proves that a given program is Lipschitz-continuous for *some* Lipschitz constant. In contrast, our work produces *exact* sensitivities for unbounded loops and we characterize a class of loops for which our technique is complete.

Recurrences in Program Analysis. Recurrence equations are a common tool in program analysis. The work of [28,29] first introduced the idea of using linear recurrences and Gröbner basis computation to synthesize loop invariants. This line of work has been further generalized in [25,22] to support more general recurrences. In [18,24] the authors apply linear recurrences to more complex programs and combine it with over-approximation techniques. The work [11] combines recurrence techniques with template-based methods to analyze recursive procedures. Recurrence equations were first used for the analysis of probabilistic loops in [8] to synthesize so-called *moment-based invariants*. This approach was further generalized by [26]. Our technique of *sensitivity recurrences* is applicable to loops whose variables’ moments do not satisfy linear recurrences. The recent work [2] studies the synthesis of invariants for such loops, but does not address sensitivity analysis.

6 Conclusion

We establish a fully automatic and exact technique to compute the sensitivities of higher moments of program variables for probabilistic loops. Our method is applicable to probabilistic loops with potentially uncountable state spaces, complex control flow, polynomial assignments, and drawing from common probability distributions. For admissible loops, we utilize closed-forms of the variables’ moments obtained through linear recurrences. Moreover, we propose the notion of *sensitivity recurrences* enabling the sensitivity analysis for probabilistic loops whose moments do not admit closed-forms. We characterize a class of loops for which we prove *sensitivity recurrences* to be sound and complete. Our experiments demonstrate the feasibility of our techniques on challenging benchmarks.

Acknowledgements This research was supported by the Vienna Science and Technology Fund WWTF 10.47379/ICT19018 grant ProbInG, the ERC Consolidator Grant ARTIST 101002685, the Austrian FWF project W1255-N23, and the SecInt Doctoral College funded by TU Wien.

References

1. Aguirre, A., Barthe, G., Hsu, J., Kaminski, B.L., Katoen, J., Matheja, C.: A Pre-expectation Calculus for Probabilistic Sensitivity. In: Proc. of POPL (2021). <https://doi.org/10.1145/3434333>
2. Amrollahi, D., Bartocci, E., Kenison, G., Kovács, L., Moosbrugger, M., Stankovic, M.: Solving invariant generation for unsolvable loops. In: Proc. of SAS (2022). https://doi.org/10.1007/978-3-031-22308-2_3
3. Barthe, G., Espitau, T., Fioriti, L.M.F., Hsu, J.: Synthesizing Probabilistic Invariants via Doob’s Decomposition. In: Proc. of CAV (2016). https://doi.org/10.1007/978-3-319-41528-4_3
4. Barthe, G., Espitau, T., Grégoire, B., Hsu, J., Strub, P.: Proving Expected Sensitivity of Probabilistic Programs. In: Proc. of POPL (2018). <https://doi.org/10.1145/3158145>

5. Barthe, G., Grégoire, B., Béguelin, S.Z.: Probabilistic relational hoare logics for computer-aided security proofs. In: Proc. of MPC (2012). <https://doi.org/10.1007/978-3-642-31113-0>
6. Barthe, G., Katoen, J.P., Silva, A.: Foundations of Probabilistic Programming. Cambridge University Press (2020). <https://doi.org/10.1017/9781108770750>
7. Barthe, G., Köpf, B., Olmedo, F., Béguelin, S.Z.: Probabilistic relational reasoning for differential privacy. In: Proc. of POPL (2012). <https://doi.org/10.1145/2103656.2103670>
8. Bartocci, E., Kovács, L., Stankovic, M.: Automatic Generation of Moment-Based Invariants for Prob-Solvable Loops. In: Proc. of ATVA (2019). https://doi.org/10.1007/978-3-030-31784-3_15
9. Bartocci, E., Kovács, L., Stankovic, M.: Analysis of Bayesian Networks via Prob-Solvable Loops. In: Proc. of ICTAC (2020). https://doi.org/10.1007/978-3-030-64276-1_12
10. Bartocci, E., Kovács, L., Stankovic, M.: Mora - Automatic Generation of Moment-Based Invariants. In: Proc. of TACAS (2020). https://doi.org/10.1007/978-3-030-45190-5_28
11. Breck, J., Cyphert, J., Kincaid, Z., Reps, T.W.: Templates and Recurrences: Better Together. In: Proc. of PLDI (2020). <https://doi.org/10.1145/3385412.3386035>
12. Chakarov, A., Sankaranarayanan, S.: Expectation Invariants for Probabilistic Program Loops as Fixed Points. In: Proc. of SAS (2014). https://doi.org/10.1007/978-3-319-10936-7_6
13. Chan, H., Darwiche, A.: When do numbers really matter? J. Artif. Intell. Res. (2002). <https://doi.org/10.1613/jair.967>
14. Chan, H., Darwiche, A.: Sensitivity Analysis in Bayesian Networks: From Single to Multiple Parameters. In: Proc. of UAI (2004)
15. Chou, Y., Yoon, H., Sankaranarayanan, S.: Predictive runtime monitoring of vehicle models using bayesian estimation and reachability analysis. In: Proc. of IROS (2020). <https://doi.org/10.1109/IROS45743.2020.9340755>
16. Durrett, R.: Probability: Theory and Examples. Cambridge University Press (2019). <https://doi.org/10.1017/9781108591034>
17. Everest, G., van der Poorten, A., Shparlinski, I., Ward, T.: Recurrence Sequences, Math. Surveys Monogr., vol. 104. Amer. Math. Soc., Providence, RI (2003)
18. Farzan, A., Kincaid, Z.: Compositional Recurrence Analysis. In: Proc. of FMCAD (2015)
19. Ghahramani, Z.: Probabilistic machine learning and artificial intelligence. Nature (2015). <https://doi.org/10.1038/nature14541>
20. Gretz, F., Katoen, J., McIver, A.: Prinsys - On a Quest for Probabilistic Loop Invariants. In: Proc. of QEST (2013). https://doi.org/10.1007/978-3-642-40196-1_17
21. Huang, Z., Wang, Z., Misailovic, S.: PSense: Automatic Sensitivity Analysis for Probabilistic Programs. In: Proc. of ATVA (2018). https://doi.org/10.1007/978-3-030-01090-4_23
22. Humenberger, A., Jaroschek, M., Kovács, L.: Invariant Generation for Multi-Path Loops with Polynomial Assignments. In: Proc. of VMCAI (2018). https://doi.org/10.1007/978-3-319-73721-8_11
23. Kauers, M., Paule, P.: The Concrete Tetrahedron - Symbolic Sums, Recurrence Equations, Generating Functions, Asymptotic Estimates. Springer (2011). <https://doi.org/10.1007/978-3-7091-0445-3>
24. Kincaid, Z., Cyphert, J., Breck, J., Reps, T.W.: Non-Linear Reasoning for Invariant Synthesis. In: Proc. of POPL (2018). <https://doi.org/10.1145/3158142>

25. Kovács, L.: Reasoning Algebraically About P-Solvable Loops. In: Proc. of TACAS (2008). https://doi.org/10.1007/978-3-540-78800-3_18
26. Moosbrugger, M., Stankovic, M., Bartocci, E., Kovács, L.: This is the moment for probabilistic loops. Proc. ACM Program. Lang. (OOPSLA2) (2022). <https://doi.org/10.1145/3563341>
27. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press (1995). <https://doi.org/10.1017/cbo9780511814075>
28. Rodríguez-Carbonell, E., Kapur, D.: Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations. In: Gutierrez, J. (ed.) Proc. of ISSAC (2004). <https://doi.org/10.1145/1005285.1005324>
29. Rodríguez-Carbonell, E., Kapur, D.: Generating All Polynomial Invariants in Simple Loops. J. Symb. Comput. (2007). <https://doi.org/10.1016/j.jsc.2007.01.002>
30. Selyunin, K., Ratasich, D., Bartocci, E., Islam, M.A., Smolka, S.A., Grosu, R.: Neural programming: Towards adaptive control in cyber-physical systems. In: Proc. of CDC (2015). <https://doi.org/10.1109/CDC.2015.7403319>
31. Stankovic, M., Bartocci, E., Kovács, L.: Moment-Based Analysis of Bayesian Network Properties. Theor. Comput. Sci. (2022). <https://doi.org/10.1016/j.tcs.2021.12.021>
32. Vasilenko, E., Vazou, N., Barthe, G.: Safe couplings: Coupled refinement types. In: Proc. of ICFP (2022). <https://doi.org/10.1145/3547643>
33. Wang, P., Fu, H., Chatterjee, K., Deng, Y., Xu, M.: Proving Expected Sensitivity of Probabilistic Programs with Randomized Variable-Dependent Termination Time. In: Proc. of POPL (2020). <https://doi.org/10.1145/3371093>

A Appendix: Probabilistic Loop Syntax

$lop \in \{and, or\}$, $cop \in \{=, \neq, <, >, \geq, \leq\}$, $Dist \in \{\text{Bernoulli}, \text{Normal}, \text{Uniform}, \dots\}$
 $\langle sym \rangle ::= a \mid b \mid \dots \mid \langle var \rangle ::= x \mid y \mid \dots$
 $\langle const \rangle ::= r \in \mathbb{R} \mid \langle sym \rangle \mid \langle const \rangle (+ \mid * \mid /) \langle const \rangle$
 $\langle poly \rangle ::= \langle const \rangle \mid \langle var \rangle \mid \langle poly \rangle (+ \mid - \mid *) \langle poly \rangle \mid \langle poly \rangle^{**n}$
 $\langle assign \rangle ::= \langle var \rangle = \langle assign_right \rangle \mid \langle var \rangle , \langle assign \rangle , \langle assign_right \rangle$
 $\langle categorical \rangle ::= \langle poly \rangle (\{ \langle const \rangle \} \langle poly \rangle)^* [\{ \langle const \rangle \}]$
 $\langle assign_right \rangle ::= \langle categorical \rangle \mid \text{Dist}(\langle const \rangle^*)$
 $\langle bexpr \rangle ::= \text{true} (\star) \mid \text{false} \mid \langle poly \rangle \langle cop \rangle \langle poly \rangle \mid \text{not} \langle bexpr \rangle \mid \langle bexpr \rangle \langle lop \rangle \langle bexpr \rangle$
 $\langle ifstmt \rangle ::= \text{if} \langle bexpr \rangle : \langle statems \rangle (\text{else if} \langle bexpr \rangle : \langle statems \rangle)^* [\text{else} : \langle statems \rangle] \text{end}$
 $\langle statem \rangle ::= \langle assign \rangle \mid \langle ifstmt \rangle \quad \langle statems \rangle ::= \langle statem \rangle^+$
 $\langle loop \rangle ::= \langle statem \rangle^* \text{while} \langle bexpr \rangle : \langle statems \rangle \text{end}$

Fig. 2: Grammar describing the syntax of probabilistic loops $\langle loop \rangle$. [26]

B Appendix: Variable Dependency

When we refer to \xrightarrow{N} as the transitive closure of \rightarrow where at least one of the direct dependencies is non-linear we mean the following. We say that $x \xrightarrow{N} y$ if

- either $x \xrightarrow{N} y$, or
- there exists some z such that either
 - $x \rightarrow z$ and $z \xrightarrow{N} y$, or
 - $x \xrightarrow{N} z$ and $z \rightarrow y$

Definition 11 (Variable Dependency). Let \mathcal{P} be a probabilistic loop and $x, y \in \text{Vars}(\mathcal{P})$. We say that x depends directly on y , in symbols $x \rightarrow y$, if y appears in an assignment of x or an assignment of x occurs in an if-statement where y appears in the if-condition. Furthermore, we say that the dependency is non-linear, in symbols $x \xrightarrow{N} y$, if y appears non-linearly in an assignment of x .

Moreover, the transitive closure \rightarrow of \rightarrow is the smallest relation satisfying:

- If $x \rightarrow y$, then $x \rightarrow y$, and
- if $x \rightarrow y \wedge y \rightarrow z$, then $x \rightarrow z$.

The relation \xrightarrow{N} formalizing transitive non-linear dependencies is the smallest relation satisfying:

- If $x \xrightarrow{N} y$, then $x \xrightarrow{N} y$,
- if $x \xrightarrow{N} y \wedge y \rightarrow z$, then $x \xrightarrow{N} z$, and
- if $x \rightarrow y \wedge y \xrightarrow{N} z$, then $x \xrightarrow{N} z$.

Definition 12 (p -Influenced Dependency). Let \mathcal{P} be a program with parameter $p \in \text{Params}(\mathcal{P})$ and $x, y \in \text{Vars}(\mathcal{P})$ with $x \rightarrow y$. Then, the direct dependency between x and y is p -influenced, in symbols $x \rightarrow_p y$ if at least one of the following conditions hold:

- An assignment of x contains y and occurs in an if-statement with the if-condition involving p or a p -dependent variable.
- An assignment of x contains y and is a probabilistic choice with some probability of the choice depending on p .
- An assignment of x contains a term $c \cdot M \cdot y$ where c is constant and M is a monomial in program variables (possibly containing y). Moreover, either c contains p or M contains a p -dependent variable.

If $x \rightarrow y$, we write $x \rightarrow_p y$ if at least one of the dependencies from x to y is p -influenced. That means $x \rightarrow_p y$ if and only if

- $x \rightarrow_p y$, or
- $x \rightarrow_p v \rightarrow y$ for some variable v , or
- $x \rightarrow v \rightarrow_p y$ for some variable v , or
- $x \rightarrow v_1 \rightarrow_p v_2 \rightarrow y$ for some variables v_1, v_2 .

If $x \rightarrow y$ and $x \not\rightarrow_p y$ we call the dependency between x and y p -free.

C Appendix: Moment Recurrences

For programs of our programming model (cf. Figure A) and a monomial M in program variables, a moment recurrence for M equates the expected value of M at iteration $n+1$, that is $\mathbb{E}(M_{n+1})$ to expected values of program variable monomials at iteration n (Definition 2). It is always possible to construct a moment recurrence for M if all variables in all branching conditions are finite valued [26]. For completeness, we restate the process introduced in [26] on how moment recurrences are constructed.

Normalized Programs. To simplify the construction of moment recurrences, we can restrict ourselves to probabilistic loops satisfying the following conditions:

- All distribution parameters are constant.
- The loop body is a flat sequence of guarded assignments. That means, every assignment is of the form $x = e_1[C]e_2$, where x is a program variable and C is a boolean condition over program variables. The expression e_1 is either a distribution or a probabilistic choice of polynomials and is assigned to the variable x if C evaluates to *true*. The expression e_2 is a single variable and is assigned to x if C evaluates to *false*.

- In the loop body, every program variable is only assigned once.

Programs satisfying these conditions are called *normalized*. Every program from our program model can be transformed into a normalized program while maintaining the joint distribution of program variables as well as the dependencies between program variables as shown in [26].

Construction of Moment Recurrences. Let \mathcal{P} be a normalized program with all program variables in branching conditions (or guards) being finitely-valued. Given a monomial in program variables M , the moment recurrence for M is constructed by starting with the expression $\mathbb{E}(M_{n+1})$ and replacing variables contained in the expression by their assignments bottom-up as they appear in the loop-body of \mathcal{P} . Throughout, the linearity of expectation is used to convert expected values of polynomials into expected values of monomials. Because \mathcal{P} is a normalized program, its loop body is a sequence of guarded assignments. Assume x is a program variable appearing in $\mathbb{E}(M_{n+1})$. Therefore, $M = M' \cdot x_{n+1}^k$ for some monomial M' free of x . The assignment of x is either a probabilistic choice of polynomials,

$$x = a_0\{p_0\} \dots \{p_{i-1}\} a_i [C] d,$$

for polynomials a_0, \dots, a_i and constant probabilities p_0, \dots, p_{i-1} , or the assignment of x is drawing from a known distribution,

$$x = \text{Dist } [C] d.$$

Because \mathcal{P} is a normalized program, C is a boolean condition and d is a program variable. In case the assignment of x is a (guarded) probabilistic choice of polynomials, $\mathbb{E}(M_{n+1})$ is rewritten to

$$\mathbb{E}(M_{n+1}) = \mathbb{E}(M' \cdot x_{n+1}^k) = \mathbb{E} \left(M' \left(d[-C] + \sum p_i a_i^k [C] \right) \right).$$

The second option is that the assignment of x is a (guarded) draw from a distribution. In this case $\mathbb{E}(M_{n+1})$ is rewritten to

$$\mathbb{E}(M_{n+1}) = \mathbb{E}(M' \cdot x_{n+1}^k) = \mathbb{E}(M' d[-C]) + \mathbb{E}(M' [C]) \mathbb{E}(\text{Dist}^k).$$

In the above expressions $[C]$ denotes the Iverson bracket, evaluating to 1 if C holds and to 0 otherwise. By assumption, all program variables in branching conditions, and hence all variables in C , are finitely-valued. Therefore, we can replace all occurrences of $[C]$ and $[-C]$ in the above equations with polynomials over variables occurring in the condition C as described in [26]. Throughout the process, the linearity of expectation is used to turn expected values of polynomials into linear combinations of expected values of monomials. By applying the replacement of program variables by their assignments for every variable bottom-up as they appear in the loop body, we end up with the moment recurrence for $\mathbb{E}(M_{n+1})$.

D Appendix: Proof of Theorem 2

Theorem 2 (Non-Admissible Sensitivities). *Let \mathcal{P} be a probabilistic program, $p \in \text{Params}(\mathcal{P})$, $x \in \text{Vars}(\mathcal{P})$, and assume all the following conditions:*

1. *All variables occurring in branching conditions are finite.*
2. *All defective variables are p -independent.*
3. *All dependencies on defective variables are p -free.*

Then, for every monomial M in program variables descending from x , Algorithm 1 terminates on input \mathcal{P} , M and p .

Proof. First, we cover the case where the monomial M contains a defective variable. If M contains a defective variable y , then y must be p -independent by condition 2. As M is a descendant of x , we have $x \rightarrow y$. Moreover, if there exists a p -dependent variable z in M different from y , Lemma 2 gives us $x \rightarrow_p y$. However, $x \rightarrow_p y$ contradicts our condition 3 that all dependencies on defective variables must be p -free. Hence, the monomial M is p -independent and Algorithm 1 terminates on line 2.

For the second, more involved case, assume all variables in M are effective and possibly p -dependent. Algorithm 1 does not terminate if and only if the algorithm adds infinitely many monomials W' to the set *Sens* on line 11 or to the set *Mom* on lines 13 or 19. Every monomial W' added to the set *Mom* occurs in the moment recurrence of W from line 16. Moreover, every monomial W' added to the set *Sens* occurs in the sensitivity recurrence of W from line 6 and hence also occurs in the moment recurrence of W . That is because the sensitivity recurrence and the moment recurrence of W share the same monomials (Definition 6).

In [26], the authors showed that every monomial W' occurring in the moment recurrence of a monomial W decreases with respect to a well-founded ordering if (A) all variables in branching conditions are finite, and (B) all variables in W and W' are effective. Premise (A) matches our condition 1. Therefore, to show that only finitely many monomials are added to *Sens* and *Mom* (and hence establish termination of Algorithm 1), it suffices to show that all monomials W' added to *Sens* and *Mom* only contain effective variables.

First, note that every monomial W' added to *Sens* or *Mom* is a descendant of the variable x . This holds because the algorithm starts with *Sens* = $\{M\}$, *Mom* = \emptyset , the monomial M is a descendant of x , and W' occurs in the moment recurrence of some $W \in \text{Sens} \cup \text{Mom}$.

Claim: All monomials W' added to Sens on line 11 only contain effective variables. Towards a contradiction, assume some monomial W' is added to *Sens* on line 11 and W' contains a defective variable y . By condition 2, y is p -independent. By Lemma 2 and condition 3, all variables in W' are p -independent. Hence, the monomial W' is p -independent and $\partial_p \mathbb{E}(W'_n)$ was replaced by 0 on line 8. Therefore, W' could not have been added to *Sens* on line 11.

Claim: All monomials W' added to Mom on line 13 only contain effective variables. Towards a contradiction, assume some monomial W' is added to *Mom*

on line 13 and W' contains a defective variable y . The monomial W' occurs in the sensitivity recurrence of W (fixed at line 6) with coefficient $(\partial/\partial_p c)$. Therefore, W' occurs in the moment recurrence of W with coefficient c . By Lemma 2 and condition 3, the constant c does not contain the parameter p . Hence, $(\partial/\partial_p c) = 0$ and $\mathbb{E}(W'_n)$ was replaced by 0 on line 9. Therefore, W' could not have been added to *Mom* on line 13.

*Claim: All monomials W' added to *Mom* on line 19 only contain effective variables.* First, note that for all monomials W' added to *Mom* on line 13, the corresponding coefficient $(\partial/\partial_p c) \neq 0$ and hence c must contain the parameter p . Therefore, by Lemma 2, for all variables y in all monomials W' added to *Mom* in the first while-loop, we have $x \twoheadrightarrow_p y$. By transitivity of \twoheadrightarrow_p , we get $x \twoheadrightarrow_p y$ for all variables y in all monomials W' added to *Mom* on line 19. Therefore, all W' added to *Mom* on line 19 cannot contain defective variables by condition 3. \square

E Appendix: Non-Admissible Benchmarks

<pre> u,w,x,y,z = 0,1,2,3,4 while *: z = z+p² {1/2} z+p y = y - 5*p*z w = 5*w + x² x = 5 + w+x u = x + p*z*y end </pre>	<pre> x,y,z,var = 1,2,a,0 d1,d2 = 5,3 run = -1 while *: run = 2*run + z² z = z+1 d1,d2 = d1*d2+3, d1+z x = 3*x + d2 + par²*z + run*z y = 3*(x-y) + par²*run end </pre>
(a) Non-Admissible (Fig. 1b)	(b) Non-Admissible-2
<pre> cnt,total = 0,0 x1,x2 = 1,2 y1,y2 = 0,3 z1,z2 = 1,5 while *: cnt = cnt + 1 x1 = x1² + q*x2 x2 = y1 + cnt + q y1 = r*(y1-cnt) + y2*cnt y2 = r*y1 + 5 z1 = cnt² - cnt + p*z1 z2 = z1*3 - 5*(z2-p) total = x2 + y2 + z2 end </pre>	<pre> y,x,z,cnt = 0,0,0,0 while *: x = DiscreteUniform(1,5) if x < 3: inc = Bernoulli(p1) cnt = cnt + inc else: inc = Bernoulli(p2) cnt = cnt - inc end f = DiscreteUniform(0,10) y = y² + x * f z = cnt² - 3*y² + x³ end </pre>
(c) Non-Admissible-3	(d) Non-Admissible-4

Fig. 3: Four parameterized non-admissible loops used for our experiments (Section 4).