TU WIEN Informatics

Moment-Based Loop Analysis

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Miroslav Stankovič, BSc
Registration Number 11836790

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr.techn. Laura Kovács
Second advisor: Univ.Prof. Dr. Ezio Bartocci

The dissertation has been reviewed by:

Ana Sokolova

Alessandro Abate

Vienna, 20th February, 2023

Miroslav Stankovič

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Miroslav Stankovič, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 20. Februar 2023

Miroslav Stankovič

To my grandfather

Acknowledgements

First of all, I would like to thank Laura Kovács for her great supervision. Laura introduced me to the area of probabilistic program analysis and guided me through the research and publishing in this area. She provided just the right amount of support to keep my PhD experience challenging but not overwhelming as I steadily joined the research community. I could not ask for a better supervisor. I would also like to thank Ezio Bartocci, my PhD co-supervisor, for the steady stream of tips, feedback, questions, and ideas. Both Laura and Ezio encouraged striving for quality, and I am incredibly grateful for their guidance and support over the years.

I am also grateful to my research group and colleagues, with whom I had many fruitful discussions and collaborations, in particular Marcel Moosbrugger, Daneshvar Amrollahi, George Kenison, Ahmad Karimi, Andrey Kofnov, and Efstathia Bura. A special thanks to Marcel - sharing an office as well as doing research with him has been excellent. I'd also like to thank Ana Sokolova and Alessandro Abate, whose feedback and suggestions helped me improve this thesis.

For the much-needed distractions from research, I would like to thank my friends and teammates, in particular Krisztina Fruzsa, Davide Longo, Marko Puza, Ľubomír Vaľovský, and Lukáš Borovský. A very special thank you goes to Martina Kmecová for being by my side during this journey.

I am also immensely grateful to family, my brother František and, especially, my parents Mária and Pavel, for their unconditional love and support, despite having hardly any idea what it is I am actually doing.

We acknowledge funding from the TU Wien Doctoral College (SecInt), the FWF research projects LogiCS W1255-N23 and P 30690-N35, the WWTF ICT19-018 grant ProbInG, the ERC Consolidator Grant ARTIST 101002685, the ERC Starting Grant 2014 SYMCAR 639270, the Wallenberg Academy Fellowship 2014 TheProSE, and the FWF grants S11405-N23, S11409-N23 (RiSE/SHiNE).

Kurzfassung

In dieser Arbeit untersuchen wir die automatisierte Analyse von Schleifen probabilistischer Programme (PPs). Wir betrachten insbesondere das Finden einer quantitativen Schleifeninvariante: eine Eigenschaft einer gegebenen Schleife, die ihr Verhalten beschreibt. Schleifeninvarianten sind der Schlüssel um über Programmschleifen logisch zu Schließen. Im Zusammenhang mit probabilistischen Programmen stellen Variablen Verteilungen dar, und die Invariante muss statistische Eigenschaften dieser Verteilungen erfassen. In unserer Arbeit konzentrieren wir uns auf die Berechnung sogenannter momentbasierter Invarianten (MBIs), invariante Eigenschaften, die den Erwartungswert und höhere (und gemischte) Momente von Programmvariablen beschreiben. Obwohl es nicht möglich ist, alle Momente zu berechnen, welche die zugrunde liegende Verteilung vollständig charakterisieren würde, sind unsere Methoden in der Lage, Momente beliebiger Ordnung zu berechnen, wodurch wir die Schleifeneigenschaften relativ genau erfassen können.

Als eines der Hauptergebnisse dieser Arbeit geben wir eine Charakterisierung von Prob-solvable Loops, einer Klasse von PP-Schleifen, für die MBIs theoretisch immer berechnet werden können. Wir beschreiben auch ein vollständig automatisiertes Verfahren zum Berechnen von MBIs jeglicher Ordnung für jedes Programm dieser Klasse. Die Methode wird implementiert und anhand mehrerer anspruchsvoller Benchmarks aus der Literatur evaluiert.

Im zweiten Teil der Dissertation untersuchen wir, wie die Moment-basierte Analyse von Prob-solvable Loops und MBIs angewendet werden kann, um über verschiedene Probleme in Bayes'schen Netzwerken (BNs) zu argumentieren. Wir erweitern das zuvor eingeführte Framework, um die Codierung von BNs als PPs zu ermöglichen, und bieten auch eine Möglichkeit, mehrere Aufgaben in der BN-Analyse als Berechnung von MBIs in einem entsprechenden PP zu codieren – wie z. B. exakte Inferenz, erwartete Anzahl von Proben oder Sensitivitätsanalyse.

Im letzten Teil der Arbeit diskutieren wir kurz Erweiterungen dieser Arbeit. Wir charakterisieren die Klasse von Programmen, für die MBIs berechnet werden können (*momentberechenbare Programme*), vollständig und untersuchen, wie Wahrscheinlichkeitsverteilungen aus MBIs automatisch approximiert werden können. Wir betrachten auch nicht momentberechenbare Programme, für die wir Kombinationen von Variablen und Approximationen betrachten.

Abstract

In this thesis, we explore automated analysis of loops of probabilistic programs (PPs). We look specifically at finding a quantitative loop invariant: a property of a given loop that describes its behaviour. Loop invariants are key for reasoning about program loops. In the context of probabilistic programs, variables represent distributions, and the invariant needs to capture the statistical properties of these distributions. In our work, we focus on computing so-called moment-based invariants (MBIs), invariant properties that describe the expected value and higher (and mixed) moments of program variables. While it is not feasible to compute all moments, which would fully characterize the underlying distribution, our methods are able to compute moments of arbitrary order, hence allowing us to capture the loop properties relatively precisely.

As one of the main results in this thesis, we give a characterization of Prob-solvable loops, a class of PP loops, for which MBIs can be, in principle, always computed. We also describe a fully automated method of computing MBIs of arbitrary order for any program of this class. The method is implemented and evaluated on several challenging benchmarks from the literature.

In the second part of the thesis, we study how moment-based analysis of Prob-solvable loops and MBIs can be applied to reasoning about various challenges in Bayesian networks (BNs). We extend the framework introduced earlier to accommodate encoding BNs as PPs and also provide a way to encode several tasks in BN analysis as computing MBIs in a corresponding PP - such as exact inference, expected number of samples, or sensitivity analysis.

In the last part of the thesis, we briefly discuss extensions of this work. We fully characterize the class of programs for which MBIs can be computed (*moment-computable programs*) and investigate how to automatically estimate probability distributions from MBIs. We also consider programs that are not moment-computable, for which we look at combinations of variables and approximations.

List of Publications

The list of peer-reviewed publications I have co-authored during my PhD studies.

[BKS19] Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. Automatic generation of moment-based invariants for prob-solvable loops. In *Proc. of ATVA 2019: the 17th International Symposium on Automated Technology for Verification and Analysis*, volume 11781 of *LNCS*, pages 255–276. Springer, 2019.

[BKS20b] Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. Mora - automatic generation of moment-based invariants. In *Proc. of TACAS 2020: the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 12078 of *LNCS*, pages 492–498. Springer, 2020.

[BKS20a] Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. Analysis of Bayesian networks via prob-solvable loops. In *Proc. of ICTAC 2020: the 17th International Colloquium on Theoretical Aspects of Computing*, volume 12545 of *LNCS*, pages 221–241. Springer, 2020.

[SBK22] Miroslav Stankovič, Ezio Bartocci, and Laura Kovács. Moment-based analysis of bayesian network properties. *Theoretical Computer Science*, 903:113–133, 2022.

[KMS+22b] Andrey Kofnov, Marcel Moosbrugger, Miroslav Stankovic, Ezio Bartocci, and Efstathia Bura. Moment-based invariants for probabilistic loops with non-polynomial assignments. In *Proc. of QEST 2022: Quantitative Evaluation of Systems - 19th International Conference*, volume 13479 of *Lecture Notes in Computer Science*, pages 3–25. Springer, 2022. **Best Paper Award**.

[KMS+22a] Ahmad Karimi, Marcel Moosbrugger, Miroslav Stankovic, Laura Kovács, Ezio Bartocci, and Efstathia Bura. Distribution estimation for probabilistic loops. In *Proc. of QEST 2022: Quantitative Evaluation of Systems - 19th International Conference*, volume 13479 of *Lecture Notes in Computer Science*, pages 26–42. Springer, 2022.

[MSBK22] Marcel Moosbrugger, Miroslav Stankovic, Ezio Bartocci, and Laura Kovács. This is the moment for probabilistic loops. *Proc. ACM Program. Lang.*, 6(OOPSLA2):1497–1525, 2022.

[ABK+22] Daneshvar Amrollahi, Ezio Bartocci, George Kenison, Laura Kovács, Marcel Moosbrugger, and Miroslav Stankovic. Solving invariant generation for unsolvable loops. In *Static Analysis - 29th International Symposium, SAS 2022*, volume 13790 of *Lecture Notes in Computer Science*, pages 19–43. Springer, 2022. **Radhia Cousot Young Researcher Best Paper Award**.

Contents

CHAPTER 1

Introduction

The recent rise in prominence of probabilistic programs (PPs) due to the emerging applications in AI and machine learning poses a significant challenge to software verification and automated reasoning. Probabilistic programs, unlike their non-probabilistic counterpart, allow drawing random values from a predefined probability distribution. The presence of probabilistic behaviour precludes treating program variables as having a certain value; instead, program variables must be treated as probabilistic distributions. This causes reasoning about general probabilistic programs to be hard [KKM19] and usually requires us to make certain simplifications, impose restrictions on the program syntax, or limit the type of analysis that can be done. Just as in the case of non-probabilistic programs, loops in PPs provide an additional challenge for program analysis.

Probabilistic programs, as used these days, can be understood in two different ways. (1) Programs encoding randomized algorithms, meant to be run, using randomness to enrich the program. (2) Programs capturing stochastic (generative) models, encoding complex probability distributions.

PPs as randomized algorithms Randomization can be leveraged to improve the performance of a program. In the *Randomized quicksort*, for example, the expected worst-case is $\mathcal{O}(n \log n)$, compared to the worst case $\mathcal{O}(n^2)$ of quicksort.

In Monte Carlo algorithms, the improvement in computation time is traded for precision. The *Miller-Rabin algorithm* [Rab80], for instance, checks whether a given number is likely to be prime. The algorithm runs in $\mathcal{O}(k \log^3 n)$ and returns a false positive with probability at most 4^{-k}, where the parameter k is the number of iterations of the algorithm.

In some cases, the addition of randomness can solve problems that would be otherwise unsolvable, as in the case of certain consensus protocols [FLP85]. Random choice also plays a crucial role in breaking the symmetry in *Hermann's self-stabilization protocol* [Her90]. Randomness is also used in the implementation of cryptographic [BGB12] and privacy [BKOB12] protocols.

Probabilistic program analysis is necessary to reason about randomized algorithms, whether it is to assess the expected runtime, estimate the probability of correct output, or quantify the uncertainty.

PPs as probabilistic models The use of randomness and probabilistic behaviour in modeling is a natural way of capturing the uncertainties present in a real-world system. Probabilistic models can represent complex probability spaces in an intuitive and compact way. Traditionally, probabilistic models had the form of a probabilistic graphical model (PGM) or a system of equations and formulas. Bayesian networks (BNs), for example, are a class of PGMs represented by an acyclic graph, modeling conditional dependence between state variables. One of the main challenges arising from probabilistic models, and Bayesian networks in particular, is making inferences based on observations or evidence.

With the recent surge in AI and machine learning development, many engineers found themselves using probabilistic models [Gha15]. As a result, using (probabilistic) programs as models is becoming increasingly more common. For many, programs are more easily understood and worked with while still allowing for mathematical precision. Program representation is particularly useful to capture generative (probabilistic) models, where programs naturally capture the steps leading to a particular distribution, generating a new sample or data point. Probabilistic languages like FIGARO[Pfe09], WEBPPL[GS14], STAN[CGH+17], INFER.NET[MWG+18], or SCENIC[FDG+19] make it easier to encode probabilistic models and capture the complex underlying distributions. One may also equip programs with symbolic parameters and seek to optimize the parameter values.

Formal analysis of probabilistic programs, when viewed as models, can help us understand the complex underlying distribution, answer inference queries, or even optimize the parameters.

1.1 Problem Statement and Motivation

Analysis of probabilistic programs is especially challenging due to its naturally quantitative character. Reasoning about problems such as variable distribution upon termination, probability of termination, expected runtime, and reachability probabilities requires dealing with quantities and functions. As in the case of non-probabilistic programs, the presence of loops and recursive behaviour proves to be the key challenge in automated PP analysis. Exact analysis is typically unfeasible due to the arbitrary, even random, number of loop executions, and even generating simple non-trivial loop properties is very tricky.

A standard approach in loop analysis involves so-called *inductive loop invariants* - properties that are true before the loop execution and that remain true after each execution of the loop body. Invariants often have a form of a logical or algebraic expression. When reasoning about a program, such invariants can replace the loop to make the reasoning easier. Invariants, however, do not, in general, uniquely represent the loop, as there may be other programs that satisfy the same invariant property and thus give an overapproximation of the loop behaviour.

To make automated invariant generation possible, invariant generation techniques impose certain restrictions, such as using templates and user guidance to generate invariants, limiting search space to a subclass of possible invariants, such as linear or polynomial functions, restricting program syntax, dealing only with finite or discrete systems, or computing approximates of or bounds on the desired values.

A variety of approaches have been considered for invariant generation. One of the earlier works [KMMM10] uses constraint-solving approach and linear templates to generate invariants in the framework of [MM05]. A martingales-based approach is used in [BEFH16] to generate invariants over expected values, further refined in [KUH19] to reason about higher moments for runtime approximation. For polynomial invariants generation, Lagrange interpolation is used in [CHWZ15] and Stengle's Positivstellensatz in [FZJ+17].

All the above approaches, however, rely heavily on templates and/or user-provided guidance. In our work, we focus on full automation for generating non-linear invariants. More specifically, the central theme of this thesis revolves around using expected value and higher (and mixed) moments as a way to capture the distribution properties of variables in probabilistic loops. We refer to the invariants over the (higher and mixed) moments of program variables as *moment-based invariants (MBIs)*.

We also explore the implications of being able to compute arbitrary MBIs on analysis of probabilistic models, in particular Bayesian networks.

As such, the work in this thesis provides the first algorithmic approaches to solving the following research goals.

Research Goal 1: Develop a fully automated approach to capture moment-based properties of probabilistic loops.

To this end, we use statistical properties to eliminate the probabilistic behaviour from the loop. Program variables, which are essentially probability distributions, can be viewed as sequences of moments. Mixed moments then capture dependencies between program variables. We consider the moments to be variables on their own and refer to them as $\mathbb{E}$-*variables*. Probabilistic updates over program variables then become non-probabilistic updates with respect to the $\mathbb{E}$-variables. Being able to reason about an arbitrary (finite) number of moments allows us to preserve as much information as desired.

With probabilistic behaviour out of the way, we adopt some of the algebraic approaches from the analysis of non-probabilistic programs. A prominent method of generating invariants in the non-probabilistic setting is based on summarizing the loop through (linear) recurrences [Kov08, KCBR17, HJK18b]. The recursive behaviour of $\mathbb{E}$-variables can be captured by a system of linear recurrences (over the $\mathbb{E}$-variables), and the closed-form expressions can be computed. The closed-form solutions then give rise to invariants involving higher and mixed moments, the *moment-based invariants* (MBIs).

The theoretical aspects of this approach, as well as the class of *Prob-solvable loops* for which this analysis can be conducted, are presented in Chapter 3, while the implementation and experiments

are covered in Chapter 4.

Research Goal 2: Apply moment-based analysis to reason about Bayesian networks.

We represent Bayesian networks as PPs, and specifically as PPs that can be analyzed using the moment-based approach developed in Chapter 3. For this, we extend the class of Prob-solvable loops and give an algorithm to convert a variety of BNs (such as discrete, Gaussian, conditional linear Gaussian, and some dynamic BNs) to a Prob-solvable loop.

A number of problems arising in BNs can then be reformulated as a task of computing moments, or MBIs, such as exact inference, sensitivity analysis, filtering, or computing the expected numbers of rejected samples in sampling-based procedures.

In Chapter 5, we cover the details of representing BNs as programs as well as using MBIs to reason about BNs.

Summary of moment-based approach To solve the main goals of this thesis, we leverage statistical methods and algebraic techniques from program analysis to fully automatically generate moment-based loop invariants and even address various problems from BN analysis. The key aspects of our approach can be summarized as follows:

- *Full automation:* no need for user guidance and/or templates,
- *Symbolic computation:* capture entire classes of programs with a single static analysis,
- *Rich model:* allow polynomial dependencies and unbounded continuous variables,
- *Theoretical foundations:* provide computational guarantees for a well-defined class of probabilistic programs,
- *Higher moments:* compute higher and mixed moment properties, in principle of arbitrary order, to more precisely capture program properties,
- *Applications:* model Bayesian networks and solve Bayesian network challenges via moment-based analysis.

The main limitation, as shall be clear by the end of this thesis, is that only a restricted class of programs can be analysed. This restriction, however, is a necessary one to guarantee computability. When higher-order moments are required, the number of $\mathbb{E}$-variables, and thus the size of the recurrence system, can grow exponentially in the worst case. Thus scalability of the approach also remains a challenge.

1.2 Related Work

In the context of probabilistic programs (PPs), formal semantics for PPs were first introduced in [Koz81], together with a deductive calculus to reason about the expected running time of

PPs [Koz85]. This approach was further refined and extended in [MM05] by introducing the weakest pre-expectations based on the weakest precondition calculus of [Dij75]. While [MM05] infers quantitative invariants only over the expected values of program variables, our moment-based invariants yield quantitative invariants over arbitrary higher-order moments, including expected values. Further, the setting of [MM05] considers PPs where the stochastic inputs are restricted to discrete distributions with finite support. However, handling Gaussian BNs requires considering also continuous distributions with infinite support, as done in our work.

Invariant synthesis with templates and hints. In [MM05], a deductive approach, the *weakest pre-expectation calculus*, for reasoning about PPs with discrete program variables is introduced. Based on the weakest pre-expectation calculus, [KMMM10] presents the first template-based approach for generating linear quantitative invariants for PPs. For this, the loop is annotated with linear template invariants, and a constraint solver is used to find template parameters that yield an invariant. Other works [CHWZ15, FZJ+17] also address the synthesis of polynomial invariants. Constraint-solving approach aided by multivariate Lagrange interpolation is used in [CHWZ15], while [FZJ+17] employs Stengle's Positivstellensatz and a transformation to a sum-of-squares problem. In [BEFH16], martingales are used to synthesize loop invariants over expected values. This approach was adapted in [KUH19] to allow reasoning about higher moments to approximate runtimes of randomized programs. All of these works [CHWZ15, FZJ+17, BEFH16, KUH19] target a slightly different problem and, unlike our approach, rely on templates. The first data-driven technique for invariant generation for PPs is presented in [BTP+22].

Probabilistic model checking A common approach for the analysis of probabilistic programs is based on probabilistic model checking [BK08]. However, this approaches [KNP11, DJKV17, KZH+11] cannot yet handle unbounded and real variables. Furthermore, probabilistic model checking tools, however, have no support for invariant generation. Our techniques could potentially aid these tools in the presence of loops.

Recurrences in non-probabilistic loops Using recurrence equations to extract quantitative invariants of loops is a well-studied technique for non-probabilistic programs [BCKR20, FK15, HJK17, HJK18b, KBCR19, Kov08, dOBP16, RCK04].

Statistical approaches A different approach to characterize the distributions of program variables is to use statistical methods such as Monte Carlo and hypothesis testing [YS06]. Simulations are, however, performed on a chosen finite number of program steps and do not provide guarantees over a potentially infinite execution, such as unbounded loops, limiting thus their use (if at all) for invariant generation.

Computing lower and upper bounds on (higher) moments Another class of related problems is finding bounds over the expected values [BGP+16, Kar94, CFGG20] and higher moments [KUH19, WHR21]. In [BGP+16], the authors also consider bounds over higher-order moments and consider nonlinear terms using interval arithmetic at the price of producing very

conservative bounds. Our approach, in contrast, natively supports probabilistic polynomial assignments and provides a precise symbolic expression over higher-order moments.

Termination Determining whether a program terminates or simply the halting problem is a well-known undecidable problem in program analysis. In the case of *probabilistic* programs, termination is a more nuanced problem. It is natural to ask not only whether a program terminates (with probability 1), known as AST, or almost sure termination, but also what is the probability of termination and whether the expected runtime of an almost surely terminating program is finite (PAST, or positive almost sure termination). Determining AST and PAST has been studied quite extensively, usually by constructing ranking supermartingales [AGR21, ACN17, CS13, CFNH16, CGMZ22, CH20, FH15, LG19, MMKK17, MBKK21]. A variety of approaches have been used to find suitable supermartingales, including reduction to constraint solving, algebraic techniques, or techniques from machine learning.

Expected runtime and cost expectations Related to termination are the problems of computing expected runtime and expected cost, or resource consumption, of probabilistic programs. The analysis is typically restricted to computing the bounds on the expected cost (or runtime) [BKKV15, KUH19, NCH18, WFG+19], with bounds on the higher moments considered in [WHR21]. A common approach [AMS20, BKK+23, KKMO16, MHG21, OKKM16] is to adapt or generalize the weakest precondition-like calculus of [MM05].

Bayesian Network analysis via PPs. To the best of our knowledge only [BKKM18] targets BNs explicitly on the source code level, by using the weakest precondition calculus similar to [KKMO16, MM05]. The PPs addressed in [BKKM18] are expressed in the *Bayesian Network Language* (BNL) fragment of the *probabilistic Guarded Command Language (pGCL)* of [MM05]. The main restriction of BNL is that loops prohibit undesired data flow across multiple loop iterations: it is not possible to assign to a variable the value of the same variable or another variable at the previous iteration. Furthermore, BNL does not natively allow to draw samples from Gaussian distribution, thus allowing only discrete BNs to be encoded in BNL. In contrast to [BKKM18], in our work we use Prob-solvable loops, as a subclass of PPs, to allow polynomial updates over random variables and parametric distribution. Variable updates of Prob-solvable loops can involve coefficients from Bernoulli, Gaussian, uniform, and other distributions, whereas variable updates drawn from Gaussian and uniform distributions can depend on other program variables. Compared to [BKKM18], we thus support reasoning about (conditional linear) Gaussian BNs and our PPs also allow data flow across loop iterations which is necessary to encode dynamic BNs.

1.3 Contributions

The work presented in this thesis pushes the state-of-the-art in automated invariant generation. Our approach is based on computing arbitrary higher-order moments of program variables, allowing more precise characterization of the underlying variable distributions.

The main parts of the thesis, covered in Chapters 3-5, are based on the following four peer-reviewed publications, of which I am the main author:

[BKS19] Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. Automatic generation of moment-based invariants for prob-solvable loops. In *Proc. of ATVA 2019: the 17th International Symposium on Automated Technology for Verification and Analysis*, volume 11781 of *LNCS*, pages 255–276. Springer, 2019.

[BKS20b] Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. Mora - automatic generation of moment-based invariants. In *Proc. of TACAS 2020: the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 12078 of *LNCS*, pages 492–498. Springer, 2020.

[BKS20a] Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. Analysis of Bayesian networks via prob-solvable loops. In *Proc. of ICTAC 2020: the 17th International Colloquium on Theoretical Aspects of Computing*, volume 12545 of *LNCS*, pages 221–241. Springer, 2020.

[SBK22] Miroslav Stankovič, Ezio Bartocci, and Laura Kovács. Moment-based analysis of bayesian network properties. *Theoretical Computer Science*, 903:113–133, 2022.

The key ideas behind the approach are introduced in [BKS19] (covered in Chapter 3) and form the theoretical foundation of this work. Technical implementation and the tool MORA are discussed in [BKS20b] and covered in Chapter 4. Applications of this work to the analysis of Bayesian networks [BKS20a, SBK22] are covered in Chapter 5.

Further results, building on the results of Chapters 3-5, are briefly discussed in Chapter 6. These results are based on the following four peer-reviewed publications:

[MSBK22] Marcel Moosbrugger, Miroslav Stankovic, Ezio Bartocci, and Laura Kovács. This is the moment for probabilistic loops. *Proc. ACM Program. Lang.*, 6(OOPSLA2):1497–1525, 2022.

[ABK+22] Daneshvar Amrollahi, Ezio Bartocci, George Kenison, Laura Kovács, Marcel Moosbrugger, and Miroslav Stankovic. Solving invariant generation for unsolvable loops. In *Static Analysis - 29th International Symposium, SAS 2022*, volume 13790 of *Lecture Notes in Computer Science*, pages 19–43. Springer, 2022. **Radhia Cousot Young Researcher Best Paper Award**.

[KMS+22b] Andrey Kofnov, Marcel Moosbrugger, Miroslav Stankovic, Ezio Bartocci, and Efstathia Bura. Moment-based invariants for probabilistic loops with non-polynomial assignments. In *Proc. of QEST 2022: Quantitative Evaluation of Systems - 19th International Conference*, volume 13479 of *Lecture Notes in Computer Science*, pages 3–25. Springer, 2022. **Best Paper Award**.

[KMS+22a] Ahmad Karimi, Marcel Moosbrugger, Miroslav Stankovic, Laura Kovács, Ezio Bartocci, and Efstathia Bura. Distribution estimation for probabilistic loops. In *Proc. of QEST 2022: Quantitative Evaluation of Systems - 19th International Conference*, volume 13479 of *Lecture Notes in Computer Science*, pages 26–42. Springer, 2022.

The former four publications, of which I am the main author, were written together with my PhD supervisors, Laura Kovács and Ezio Bartocci. Their experience, detailed feedback, and revisions played a major role in shaping these publications (this is also true for the publications covered next).

The latter quadruple of publications is a result of a broader collaboration with my colleagues. Here, I summarize, as required, my relative contributions insofar as can be identified and to the best of my judgment. The research covered in [MSBK22] was conducted together with Marcel Moosbrugger. Of our contributions, roughly equally split between us, my focus was mostly on the more theoretical part of moment-computability. The work of [ABK+22] arose from a discussion we had together about finding combinations of variables to which our previously developed methods could be applied. I was mostly involved in the earlier stages of formulating the problem and forming the initial approach/algorithm for the synthesis of variable combinations. In [KMS+22b], we apply the PCE to approximate non-polynomial functions to get a Prob-solvable loop. The restrictions to guarantee that the approximations lead to a Prob-solvable loop were formulated by Marcel and me and adapted by Andrey, who also came up with how to use the PCE within PPs. The idea of using MBIs to estimate a distribution arose naturally in the discussions, and we addressed it properly in [KMS+22a]. I was involved in applying the two estimation methods to PPs, as well as designing the tests to evaluate the estimations.

Summary of contributions We summarize our contributions below.

- We introduce the class of Prob-solvable loops with probabilistic assignments over random variables and distributions in Section 3.2. The class is further extended in Section 5.2 to accommodate Bayesian network analysis.
- We show that Prob-solvable loops can be modeled as C-finite recurrences over higher-order moments of variables (Theorem 3.1).
- We provide a fully automated approach that derives moment-based invariants over arbitrary higher-order moments of Prob-solvable loops (Algorithm 1).
- We provide a sound encoding of BNs as Prob-solvable loops, in particular addressing discrete BNs (disBNs), Gaussian BNs (gBNs), conditional linear Gaussian BNs (clgBNs), and dynamic BN (dynBNs) (Section 5.3).
- We formalize several BN problems as moment-based invariant generation tasks in Prob-solvable loops (Section 5.4).
- We implemented the theoretical contributions in a tool called MORA (Chapter 4).

1.4 Outline and Organization

In Chapter 2, we introduce key results from loop analysis, algebra, probability, and statistics, which form the basis for our work.

Chapter 3, based on [BKS19], forms the foundation of this thesis. It defines the class of Prob-solvable loops (Section 3.2), establishes the notion of moment-based invariants and gives an algorithm to compute them (Section 3.3).

In Chapter 4, based on [BKS20b], we introduce tool MORA, which implements the previously discussed methods (Sections 4.2-4.4). Experiments and evaluation are discussed in Section 4.5

We explore how to use the methods developed in Chapter 3 in Bayesian network analysis in Chapter 5. In Section 5.2, the model of Prob-solvable is extended and Section 5.3 shows how BNs can be encoded as Prob-solvable loops. In Section 5.4, we discuss how BN analysis can be done via reasoning about Prob-solvable loops. This Chapter is based on [BKS20a, SBK22].

Chapter 6 briefly presents further developments and generalizations, building on the work of Chapters 3-5. The results presented here come from [ABK+22, KMS+22a, KMS+22b, MSBK22].

The thesis is concluded in Chapter 7.

CHAPTER 2

Preliminaries

In this chapter, we cover background material that forms a foundation of the main results in Chapters 3-5.

Throughout this paper, let $\mathbb{N}, \mathbb{Z}, \mathbb{R}$ denote the set of natural, integer, and real numbers. We use $\mathbb{E}$ for the expectation operator.

2.1 Probabilities, Expectation, and Moments

We first introduce basic notions from probability and statistics in order to reason about probabilistic systems and refer to [Lin92] for further details.

Definition 2.1 (σ-algebra)**.** *Given a set Ω, A σ-algebra F is a non-empty collection of subsets of Ω, such that:*

1. *Ω is in F,*
2. *if A is in F, then so is its complement $\Omega \setminus A$, and*
3. *F is closed under countable unions.*

Definition 2.2 (Measurable space)**.** *Measurable space the tuple (Ω, F) consisting of a set Ω and a σ-algebra F on Ω.*

Definition 2.3 (Probability space)**.** *A* probability space *is a triple (Ω, F, P) consisting of a sample space Ω denoting the set of outcomes, where $\Omega \neq \emptyset$, a σ-algebra F with $F \subset 2^{\Omega}$, denoting a set of events, and a probability measure $P : F \to [0, 1]$ s.t. $P(\Omega) = 1$.*

We now define random variables and their higher-order moments.

Definition 2.4 (Random variable)**.** *A* random variable $X : \Omega \to E$ *is a measurable function from a set* Ω *of possible outcomes (also called sample space) to a measurable space* E*. If* Ω *is finite or countable, the random variable* X *is called* discrete*; otherwise,* X *is* continuous*. For a given random variable* X*, we will denote the sample space of* X *by* $\Omega(X)$*. The probability that a random variable* X *has a value from a measurable set* $S \subset E$ *is* $P(X \in S) = P(\{w \in \Omega | X(w) \in S\})$*. For a singleton set, we write* $P(X = x)$ *for* $P(X \in \{x\})$*.*

Example 2.1. *Consider a regular dice roll. We can describe it with a probability space* (Ω, F, P)*, with* $\Omega = \{1, 2, 3, 4, 5, 6\}$*,* $F = 2^{\Omega}$*, and* $P(A) = \frac{|A|}{6}$*. We can represent the roll with a random variable* $X : \Omega \to (\Omega, F)$*, with* $X(x) = x$*. To represent whether the roll is odd or even, we let* $B = \{odd, even\}$ *and define random variable* $Y : \Omega \to (B, 2^B)$*, with* $Y(x) = odd$ *for odd* x *and* $Y(x) = even$ *for even* x*.*

When working with a random variable X, one is, in general, interested in expected values and other moments.

Definition 2.5 (Expected value)**.** *An* expected value of a random variable X *defined on a probability space* (Ω, F, P) *is the Lebesgue integral:* $\mathbb{E}[X] = \int_{\Omega} X \cdot dP$*. In the special case when* Ω *is discrete, that is the outcomes are* $X_1, \dots X_N$ *with corresponding probabilities* $p_1, \dots p_N$*, we have* $\mathbb{E}[X] = \sum_{i=1}^{N} X_i \cdot p_i$*. The expected value of* X *is often also referred to as the* average*,* mean*, or* μ *of* X*.*

Definition 2.6 (Higher-Order moments)**.** *Let* X *be a random variable,* $c \in \mathbb{R}$*, and* $k \in \mathbb{N}$*. We write* $Mom_k[c, X]$ *to denote the* kth moment about c of X*, which is defined as:*

$$Mom_k[c, X] = \mathbb{E}[(X - c)^k] \tag{2.1}$$

In this thesis, we will be primarily working with moments about 0, called *raw moments*, and about the mean $\mathbb{E}[X]$, called *central moments*. We note, though, that we can move to moments of X with different centers.

Theorem 2.1 (Transformation of center)**.** *Let* X *be a random variable,* $c, d \in \mathbb{R}$*, and* $k \in \mathbb{N}$*. The* kth moment about d of X *can be calculated from the moments about* c *of* X *by:*

$$E\left[(X - d)^k\right] = \sum_{i=0}^{k} \binom{k}{i} E\left[(X - c)^i\right] (c - d)^{k-i}.$$

Moments involving multiple random variables are called *mixed moments*.

Definition 2.7 (Mixed moments)**.** *Let* $X_1, \dots, X_m$ *be random variables and* $k_i \in \mathbb{N}$ *for* $1 \leq i \leq m$*. Then* $\mathbb{E}[X_1^{k_1} \cdots X_m^{k_m}]$ *is a mixed moment of order* $k_1 + \cdots + k_m$*.*

For arbitrary random variables X and Y, we have the following basic properties about their expected values and other moments:

- $\mathbb{E}[c] = c$ for a constant $c \in \mathbb{R}$,

- expected value is linear, $\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$ and $\mathbb{E}[c \cdot X] = c \cdot \mathbb{E}[X]$,
- expected value is not multiplicative, in general $\mathbb{E}[X \cdot Y] \neq \mathbb{E}[X] \cdot \mathbb{E}[Y]$,
- expected value is multiplicative for independent random variables.

As a consequence of the above, expected values of monomials over arbitrary random variables, e.g. $\mathbb{E}[X \cdot Y^2]$, cannot be in general further simplified.

The moments of a random variable X with bounded support fully characterise its value distribution. While computing all moments of X is generally impossible, knowing only a few moments of X gives useful information about its value distributions.

Definition 2.8 (Common moments)**.** Variance *measures how spread the distribution is and is defined as the second central moment:* $Var[X] = Mom_2[\mathbb{E}[X], X]$.

Covariance *is a mixed moment measuring the variability of two distributions and is defined as* $Cov[X, Y] = \mathbb{E}[(X - \mathbb{E}[X]) \cdot (Y - \mathbb{E}[Y])]$.

Skewness *measures the asymmetry of the distribution and is defined as the normalised third central moment:*

$$Skew[X] = \frac{Mom_3[\mathbb{E}[X], X]}{(Var[X])^{3/2}}.$$

Basic results about variance and covariance state that

$$\begin{aligned} Cov[X, X] &= Var[X], \\ Var[X] &= \mathbb{E}[X^2] - (\mathbb{E}[X]^2), \text{ and} \\ Cov[X, Y] &= \mathbb{E}[X \cdot Y] - \mathbb{E}[X] \cdot \mathbb{E}[Y]. \end{aligned}$$

Another useful tool in reasoning about probability spaces is the so-called characteristic function.

Definition 2.9 (Characteristic function)**.** *The* characteristic function *of a random variable X, denoted by $\phi_X(t)$, is the Fourier transform of its probability density function (pdf). That is, $\phi(t) = \mathbb{E}[e^{itX}]$, with a bijective relation between probability distributions and characteristic functions.*

The characteristic function $\phi_X(t)$ of a random variable X captures the value distribution induced by X. In particular, the characteristic function $\phi_X(t)$ of X enables inferring properties about distributions given by weighted sums of X and other random variables, and thus also about statistical higher-order moments of X.

Complex probability spaces are often represented using probabilistic graphical models (PGMs), most commonly Bayesian networks, or simply BNs.

Definition 2.10 (Bayesian network)**.** *A Bayesian network consists of a directed acyclic graph G and a set of conditional probability distributions.*

The nodes N_G of G represent the random variables of the model, and the edges E_G of G the causal dependencies between variables. For each $X \in N_G$ let $Par(X) = \{Y | (Y, X) \in E\}$ be the set of variables X depends on. A conditional probability distribution f_X is given for each node $X \in G$ given its parents $Par(X)$.

Conditional probability distributions in a BN can be specified in different ways, and we overview the most common ones. For a discrete variable X, dependencies are often given by a conditional probability table by listing all possible values of parent variables $Par(X)$ and the corresponding values of X. In the case of a continuous variable X, dependencies can be specified using Gaussian distributions. Another common dependency in a BN is a deterministic one when the value of a node X is determined by the values of its parents from $Par(X)$; e.g., a binary variable can be true if all its (binary) parents are true or if one of its parents is true.

Definition 2.11 (Variants of Bayesian networks)**.** *The following variants of Bayesian networks are considered throughout the paper:*

- *A* discrete Bayesian network (disBN) *is a BN whose variables are discrete-valued.*
- *A* Gaussian Bayesian network (gBN) *is a BN whose dependencies are given by the Gaussian distribution in which, for any BN node X, we have $P(X|Par(X)) = \mathcal{G}(\mu_X, \sigma_X^2)$, with $\mu_X = \alpha_X + \sum_{k=1}^{m_X} \beta_{X,k} Y_{X,k}$, $Par(X) = \{Y_1, \cdots, Y_{m_X}\}$ and σ_X^2 is fixed.*
- *A* conditional linear Gaussian Bayesian network (clgBN) *is a BN in which (i) continuous nodes X are not parents of discrete nodes Y; (ii) the local distribution of each discrete node Y is a conditional probability table (CPT); (iii) the local distribution of each continuous node X is a set of Gaussian distributions, one for each configuration of the discrete parents Y, with the continuous parents acting as regressors.*
- *A* dynamic Bayesian network (dynBN) *is a structured BN consisting of a series of time slices that represent the state of all the BN nodes X at a certain time t. For each time-slice, a dependency structure between the variables X at that time is defined by intra-time-slice edges. Additionally, there are edges between variables from different slices—inter-time-slice edges, with their directions following the direction of time.*

2.2 C-Finite Recurrences

We recall basic mathematical properties about recurrences and higher-order moments of variable values – for more details see [KP11, Lin92].

While sequences and recurrences are defined over arbitrary fields of characteristic zero, in our work, we only focus on sequences/recurrences over $\mathbb{R}$.

Definition 2.12 (Sequence)**.** *A univariate* sequence *in $\mathbb{R}$ is a function $f : \mathbb{Z} \to \mathbb{R}$. A* recurrence *for a sequence $f(n)$ is*

$$f(n+r) = R(f(n), f(n+1), \ldots, f(n+r-1), n), \qquad \text{with } n \in \mathbb{N},$$

for some function $R : \mathbb{R}^{r+1} \to \mathbb{R}$*, where* $r \in \mathbb{N}$ *is called the* order *of the recurrence.*

For simplicity, we denote by $f(n)$ both the recurrence of $f(n)$ as well as the recurrence equation $f(n) = 0$. When solving the recurrence $f(n)$, one is interested in computing a *closed form* solution of $f(n)$, expressing the value of $f(n)$ as a function of n for any $n \in \mathbb{N}$. In our work, we only consider the class of *linear recurrences with constant coefficients*, also called *C-finite recurrences*.

Definition 2.13 (C-finite recurrences)**.** *A* C-finite recurrence $f(n)$ *satisfies the linear homogeneous recurrence with constant coefficients:*

$$f(n+r) = a_0 f(n) + a_1 f(n+1) + \ldots + a_{r-1} f(n+r-1), \qquad \text{with } r, n \in \mathbb{N}, \quad (2.2)$$

where r *is the* order *of the recurrence, and* $a_0, \ldots, a_{r-1} \in \mathbb{R}$ *are constants with* $a_0 \neq 0$*.*

An example of a C-finite recurrence is the recurrence of Fibonacci numbers satisfying the recurrence $f(n+2) = f(n+1) + f(n)$, with initial values $f(0) = 0$ and $f(1) = 1$. Unlike arbitrary recurrences, closed forms of C-finite recurrences $f(n)$ always exist [KP11] and have the form

$$f(n) = P_1(n)\theta_1^n + \cdots + P_s(n)\theta_s^n, \quad (2.3)$$

where $\theta_1, \ldots, \theta_s \in \mathbb{R}$ are the distinct roots of the characteristic polynomial of $f(n)$ and $P_i(n)$ are polynomials in n. Closed forms of C-finite recurrences are called *C-finite expressions*. We note that while the C-finite recurrence (2.2) is homogeneous, inhomogeneous C-finite recurrences can always be translated into homogeneous ones, as the inhomogeneous part of a C-finite recurrence is a C-finite expression.

In our work, we focus on the analysis of Prob-solvable loops and consider loop variables x as sequences $x(n)$, where $n \in \mathbb{N}$ denotes the loop iteration counter. Thus, $x(n)$ gives the value of the program variable x at iteration n.

Theorem 2.2 (Closed-form [KP11])**.** *Every C-finite sequence* $(a_n)_{n=0}^{\infty}$ *can be written as an* exponential polynomial*, that is* $a_n = \sum_{i=1}^{m} n^{d_i} u_i^n$ *for some natural numbers* $d_i \in \mathbb{N}$ *and complex numbers* $u_i \in \mathbb{C}$*. We refer to* $\sum_{i=1}^{m} n^{d_i} u_i^n$ *as the* closed-form *or the* solution *of the sequence* $(a_n)_{n=0}^{\infty}$ *or its recurrence.*

An important fact is that closed forms of linear recurrences with constant coefficients of *any order* always exist and are computable. This also holds for all variables in *systems* of linear recurrences with constant coefficients.

CHAPTER 3

Prob-solvable Loops

This chapter is based on a joint work [BKS19] with Laura Kovács and Ezio Bartocci, published in the proceedings of ATVA 2019.

[BKS19] Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. Automatic generation of moment-based invariants for prob-solvable loops. In *Proc. of ATVA 2019: the 17th International Symposium on Automated Technology for Verification and Analysis*, volume 11781 of *LNCS*, pages 255–276. Springer, 2019.

3.1 Overview

In this chapter we introduce a *fully automated approach* to compute invariant properties over *higher-order moments* of so-called *Prob-solvable loops*, to stand for *probabilistic P-solvable loops*. Prob-solvable loops are PPs that extend the imperative P-solvable loops described in [Kov08] with probabilistic assignments over random variables and parametrised distributions. As such, variable updates are expressed by random polynomial, and not only affine, updates (see Section 3.2).

Consider, for example, the PPs of Figure 3.1(A) and Figure 3.1(B): the expected value of variable s at each loop iteration is the same in both PPs, while the variance of the value distribution of s differs in general (a similar behaviour is also exploited by Figure 3.1(C)-(D)). Thus, Figure 3.1(A) and Figure 3.1(B) do not have the same invariants over higher-order moments; yet, many approaches would fail to identify such differences and only compute expected values of variables. Our work uses statistical properties to eliminate probabilistic choices and turn random updates into recurrence relations over higher-order moments of program variables. We show that higher-order moments of Prob-solvable loops can be described by C-finite recurrences (Theorem 3.1). We further solve such recurrences to derive *moment-based invariants* of Prob-solvable loops (Section 3.3). Moment-based invariants describe statistical properties of loop variables that hold at arbitrary loop iteration, hence invariants. They are represented by closed-form

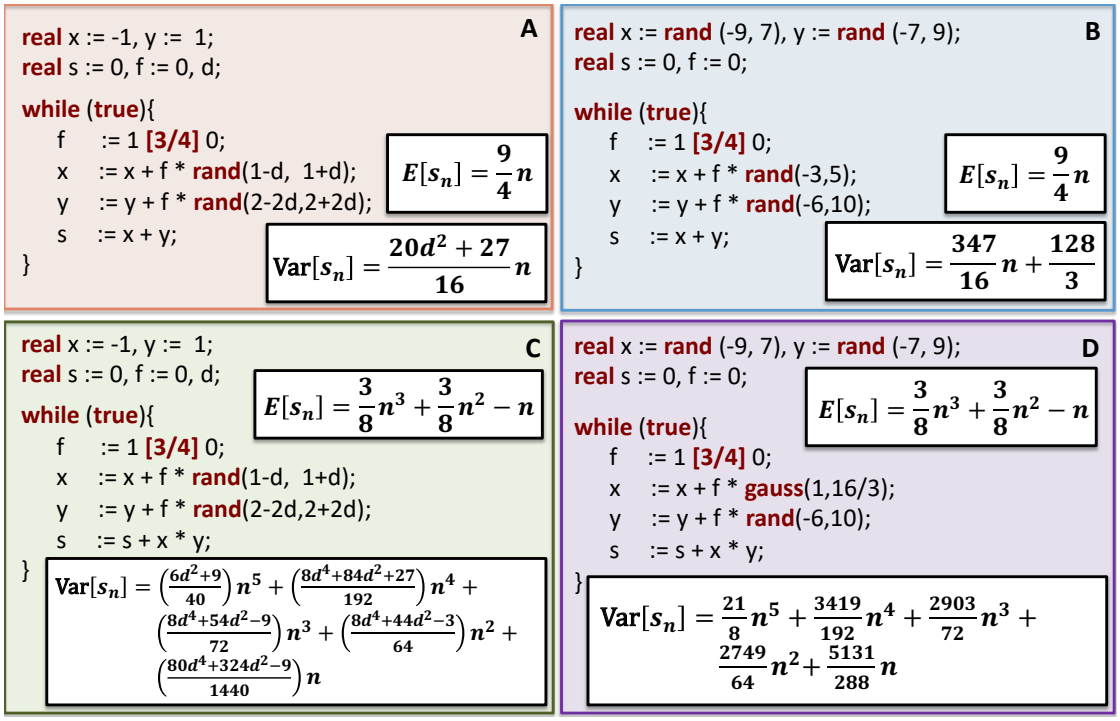

Figure 3.1: Examples of four Prob-solvable loops. `f:=1 [3/4] 0` is a statement that assigns to `f` the value 1 with probability $\frac{3}{4}$ and the value 0 with probability $1 - \frac{3}{4} = \frac{1}{4}$. The function `rand(a,b)` samples a random number from a uniform distribution with support in the real interval $[a, b]$ and the function `gauss`(μ, σ^2) samples a random number from a normal distribution with mean μ and variance σ^2. For each loop, we provide the moment-based invariants for the first ($\mathbb{E}[-]$) and second moments ($Var[-]$) of s computed using our approach, where n denotes the loop counter.

solutions for higher-order moments of program variables and capture how the statistical moments of program variables evolve through the loop iterations.

To the best of our knowledge, no other method is able to derive higher-order moments of PPs in a fully automated approach. Our work hence allows replacing, for example, the required human guidance of [GKM13, KNP11] for Prob-solvable loops. Unlike existing works, we support PPs with parametrised distributions (e.g., in Figure 3.1(A)): instead of taking concrete instances of a given parametrised distribution, we automatically infer invariants of the entire class of PPs characterised by the considered parametrised distribution.

Our approach is both sound and terminating: given a Prob-solvable loops and an integer $k \geq 1$, we automatically infer the moment-based invariants over the kth moments of our input loop (see Section 3.3). Unlike the approach of [Kov08] for deriving polynomial invariants of non-probabilistic (P-solvable) loops, our work only computes closed form expressions over higher-order moments and does not employ Gröbner basis computation to eliminate loop counters from the derived closed forms. As such, our moment-based invariants are not restrictive to polynomial

properties but are linear combinations of polynomial expressions and exponential sequences over the loop counter. Moreover, Prob-solvable loops are more expressive than P-solvable loops as they are not restricted to deterministic updates but allow random assignments over variables.

3.2 Programming Model: Prob-solvable Programs

We now introduce our programming model of *Prob-solvable programs*, to stand for *probabilistic P-solvable programs*. P-solvable programs [Kov08] are non-deterministic loops whose behaviour can be expressed by a system of C-finite recurrences over program variables. Prob-solvable programs extend P-solvable programs by allowing probabilistic assignments over random variables and distributions.

Prob-solvable loops Let $m \in \mathbb{N}$ and $x_1, \dots x_m$ denote real-valued program variables. We define Prob-solvable loops with $x_1, \dots x_m$ variables as programs of the form:

$$I; \texttt{while(true)}\{U\}, \qquad \text{where:} \tag{3.1}$$

- I is a sequence of initial assignments over $x_1, \dots, x_m$. That is, I is an assignments sequence $x_1 := c_1; x_2 := c_2; \dots x_m := c_m$, with $c_i \in \mathbb{R}$ representing a number drawn from a known distribution [1] - in particular, c_i can be a real constant.
- U is the loop body and is a sequence of m random updates, each of the form:

$$x_i := a_i x_i + P_i(x_1, \dots x_{i-1}) \; [p_i] \; b_i x_i + Q_i(x_1, \dots x_{i-1}), \tag{3.2}$$

  or, in the case of a deterministic assignment,

$$x_i := a_i x_i + P_i(x_1, \dots x_{i-1}), \tag{3.3}$$

  where $a_i, b_i \in \mathbb{R}$ are constants and $P_i, Q_i \in \mathbb{R}[x_1, \dots, x_{i-1}]$ are polynomials over program variables $x_1, \dots, x_{i-1}$. Further, $p_i \in [0, 1]$ in (3.2) is the probability of updating x_i to $a_i x_i + P_i(x_1, \dots x_{i-1})$, whereas the probability to update x_i to $b_i x_i + Q_i(x_1, \dots x_{i-1})$ in (3.2) is $1 - p_i$.

The coefficients a_i, b_i and the coefficients of P_i and Q_i in the variable assignments (3.2)-(3.3) of Prob-solvable loops can be drawn from a random distribution as long as the moments of this distribution are known and are independent of program variables $x_1, \dots, x_m$. Hence, the variable updates of Prob-solvable loop can involve coefficients drawn from Bernoulli, uniform, normal, and other distributions. Moreover, Prob-solvable loops support parametrised distributions, for example one may have the random distribution `rand(d_1,d_2)` with arbitrary $d_1, d_2 \in \mathbb{R}$ symbolic constants. Similarly, rather than only considering concrete numeric values of p_i, the probabilities p_i in the probabilistic updates (3.2) of Prob-solvable loops can also be symbolic constants.

[1] a known distribution is a distribution with known and computable moments

Example 3.1. *The programs in Fig. 3.1 are Prob-solvable, using uniform distributions given by* `rand()`*. Fig. 3.1(D) also uses normal distribution given by* `gauss()`*. Note that while the random distributions of Fig. 3.1(B,D) are defined in terms of concrete constants, Fig. 3.1(A,C) have a parametrised random distribution, defined in terms of* $d \in \mathbb{R}$*.*

Prob-solvable loops and moment-based recurrences Let us now consider a Prob-solvable program with $n \in \mathbb{N}$ denoting the loop iteration counter. We show that variable updates of Prob-solvable programs yield special recurrences in n, called *moment-based recurrences*. For this, we consider program variables $x_1, \ldots, x_m$ as sequences $x_1(n), \ldots, x_m(n)$ allowing us to precisely describe relations between values of x_i at different loop iterations. Using this notation, probabilistic updates (3.2) over x_i turn $x_i(n)$ into a random variable, yielding the relation (similarly, for deterministic updates (3.3)):

$$x_i(n+1) = a_i x_i(n) + P_i(x_1(n), \ldots, x_{i-1}(n))\ [p_i]\ b_i x_i(n) + Q_i(x_1(n), \ldots, x_{i-1}(n)).$$

The relation above could be treated as a recurrence equation over random variables $x_i(n)$ provided the probabilistic behaviour depending on p is encoded (as an extension) into a recurrence equation. To analyse such probabilistic updates of Prob-solvable loops, for each random variable $x_i(n)$ we consider their expected values $\mathbb{E}[x_i(n)]$ and create new recurrence variables from expected values of monomials over original program variables (e.g., a new variable $\mathbb{E}[x_i \cdot x_j]$). We refer to these new recurrence variables as $\mathbb{E}$-*variables*. We note that any program variable yields an $\mathbb{E}$-variable, but not every $\mathbb{E}$-variable corresponds to one single program variable as $\mathbb{E}$-variables are expected values of monomials over program variables. We now formulate recurrence equations over $\mathbb{E}$-variables rather than over program variables, yielding *moment-based recurrences*.

Definition 3.1 (Moment-based recurrences)**.** *Let $x(n)$ be a sequence of random variables. A* moment-based recurrence *for x is a recurrence over $\mathbb{E}$-variable $\mathbb{E}[x]$:*

$$\mathbb{E}[x(n+r)] = R(\mathbb{E}[x(n)], \mathbb{E}[x(n+1)], \ldots, \mathbb{E}[x(n+r-1)], n) \quad (n \in \mathbb{N}),$$

for some function $R : \mathbb{R}^{r+1} \to \mathbb{R}$, where $r \in \mathbb{N}$ is the order *of the moment-based recurrence.*

Note that variable updates $x_i := f_1(x_i)\ [p_i]\ f_2(x_i)$ yield the relation

$$\begin{array}{rcl} \mathbb{E}[x_i(n+1)] & = & \mathbb{E}[p_i \cdot f_1(x_i(n)) + (1-p_i) \cdot f_2(x_i(n))] \\ & = & p_i \cdot \mathbb{E}[f_1(x_i(n))] + (1-p_i) \cdot \mathbb{E}[f_2(x_i(n))]. \end{array} \tag{3.4}$$

Thanks to this relation, probabilistic updates (3.2) are rewritten into the moment-based recurrence equations

$$\begin{array}{rcl} \mathbb{E}[x_i(n+1)] & = & p_i \cdot \mathbb{E}[a_i x_i(n) + P_i(x_1(n), \ldots, x_{i-1}(n))] \\ & & +(1-p_i) \cdot \mathbb{E}[b_i x_i(n) + Q_i(x_1(n), \ldots, x_{i-1}(n))]. \end{array} \tag{3.5}$$

In particular, we have $\mathbb{E}[x_i(n+1)] = p_i \cdot \mathbb{E}[a_i x_i(n) + P_i(x_1(n), \ldots, x_{i-1}(n))]$ for the deterministic assignments of (3.3) (that is, $p_i = 1$ in (3.3)).

By using properties of expected values of expressions $expr_1, expr_2$ over random variables, we obtain the following simplification rules:

$$\begin{array}{lcl} \mathbb{E}[expr_1 + expr_2] & \to & \mathbb{E}[expr_1] + \mathbb{E}[expr_2] \\ \mathbb{E}[expr_1 \cdot expr_2] & \to & \mathbb{E}[expr_1] \cdot \mathbb{E}[expr_2], \text{ if } expr_1, expr_2 \text{ are independent} \\ \mathbb{E}[c \cdot expr_1] & \to & c \cdot \mathbb{E}[expr_1] \\ \mathbb{E}[c] & \to & c \\ \mathbb{E}[\mathcal{D} \cdot expr_1] & \to & \mathbb{E}[\mathcal{D}] \cdot \mathbb{E}[expr_1] \end{array} \tag{3.6}$$

where $c \in \mathbb{R}$ is a constant and $\mathcal{D}$ is a known independent distribution.

Example 3.2. *The moment-based recurrences of the Prob-solvable loop of Fig. 3.1(A) are:*

$$\begin{cases} \mathbb{E}[f(n+1)] &= \frac{3}{4}\mathbb{E}[1] + \frac{1}{4}\mathbb{E}[0]) \\ \mathbb{E}[x(n+1)] &= \mathbb{E}\big[x(n) + f(n+1) \cdot rand(1-d, 1+d)\big] \\ \mathbb{E}[y(n+1)] &= \mathbb{E}\big[y(n) + f(n+1) \cdot rand(2-2d, 2+2d)\big] \\ \mathbb{E}[s(n+1)] &= \mathbb{E}\big[x(n+1) + y(n+1)\big] \end{cases}$$

By using the simplification rules (5.4) *on the above recurrences, we obtain the following simplified moment-based recurrences of Fig. 3.1(A):*

$$\begin{cases} \mathbb{E}[f(n+1)] &= \frac{3}{4} \\ \mathbb{E}[x(n+1)] &= \mathbb{E}[x(n)] + \mathbb{E}[f(n+1)] \cdot \mathbb{E}[rand(1-d, 1+d)] \\ \mathbb{E}[y(n+1)] &= \mathbb{E}[y(n)] + \mathbb{E}[f(n+1)] \cdot \mathbb{E}[rand(2-2d, 2+2d)] \\ \mathbb{E}[s(n+1)] &= \mathbb{E}[x(n+1)] + \mathbb{E}[y(n+1)] \end{cases} \tag{3.7}$$

In Section 3.3 we show that Prob-solvable loops can further be rewritten into a system of C-finite recurrences over $\mathbb{E}$-variables.

Prob-solvable loops and mutually dependent updates Consider PP loops with mutually dependent affine updates

$$x_i := \sum_{k=1}^{m} a_{i,k} x_k \; + c_i \; [p_i] \; \sum_{k=1}^{m} b_{i,k} x_k \; + d_i, \tag{3.8}$$

where $a_{i,k}, b_{i,k}, c_i, d_i \in \mathbb{R}$ are constants. While such assignments are not directly captured by updates (3.2) of Prob-solvable loops, this is not a restriction of our work. Variable updates given by (3.8) yield mutually dependent C-finite recurrences over $\mathbb{E}$-variables. Using methods from [KP11], this coupled system of C-finite recurrences can be rewritten into an equivalent system of independent C-finite recurrences over $\mathbb{E}$-variables, yielding an independent system of moment-based recurrences over which our invariant generation algorithm from Section 3.3 can be applied. Hence probabilistic loops with affine updates are special cases of Prob-solvable loops.

Multi-path Prob-solvable loops While (5.1) defines Prob-solvable programs as single-path loops, the following class of multi-path loops can naturally be modeled by Prob-solvable programs:

$$I; \texttt{while(true)}\{\texttt{if } t \texttt{ then } U_1 \texttt{ else } U_2\}, \qquad \text{where:} \tag{3.9}$$

I is as in (5.1), t is a boolean-valued random variable, and U_1 and U_2 are respectively sequences of deterministic updates $x_i := a_i x_i + P_i(x_1, \dots x_{i-1})$ and $x_i := b_i x_i + Q_i(x_1, \dots x_{i-1})$ as in (3.3). PPs (3.9) can be rewritten to equivalent Prob-solvable loops, as follows. A pair of updates $x := u_1[p]v_1$ from U_1 and $x := u_2[p]v_2$ from U_2 is rewritten by the following sequence of updates:

$$\begin{aligned} &f := 1[p]0; \\ &g := 1[p]0; \\ &x := t(u_1 f + v_1(1-f)) + (1-t)(u_2 g + v_2(1-g)) \end{aligned} \tag{3.10}$$

with f, g fresh program variables. The resulting program is Prob-solvable and we can thus compute moment-based invariants of multi-path loops as in (3.9). The programs COUPON, RANDOM˙WALK˙2D of Table 3.1 are Prob-solvable loops corresponding to such multi-path loops.

3.3 Moment-Based Invariants of Prob-solvable Loops

Thanks to probabilistic updates, the values of program variables of Prob-solvable loops after a specific number of loop iterations are not a priory determined. The value distributions $x_i(n)$ of program variables x_i are, therefore, random variables. When analysing Prob-solvable loops, and in general probabilistic programs, one is therefore required to capture relevant properties over expected values and higher moments of the variables in order to summarise the value distribution of program variables precisely.

Moment-based invariants We are interested in automatically generating so-called *moment-based invariants* of Prob-solvable loops. Moment-based invariants are properties over expected values and higher moments of program variables, such that these properties hold at arbitrary loop iterations (and hence are invariants).

Automated generation of moment-based invariants of Prob-solvable loops Our method for generating moment-based invariants of Prob-solvable loops is summarized in Algorithm 1. Algorithm 1 takes as input a Prob-solvable loop $\mathcal{P}$ and a natural number $k \geq 1$ and returns *moment-based invariants over the kth moments* of the program variables $\{x_1, \dots, x_m\}$. We denote by n the loop counter of $\mathcal{P}$.

Theorem 3.1. *Higher-order moments of variables in Prob-solvable loops can be modeled by C-finite recurrences over $\mathbb{E}$-variables.*

Algorithm 1 Moment-Based Invariants of Prob-solvable Loops

Input: Prob-solvable loop $\mathcal{P}$ as defined in (5.1), with variables $\{x_1, \ldots, x_m\}$, and $k \geq 1$
Output: Set MI of Moment-based invariants of $\mathcal{P}$ over the kth moments of $\{x_1, \ldots, x_m\}$
Assumptions: $n \in \mathbb{N}$ is the loop counter of $\mathcal{P}$

1: Extract the moment-based recurrence relations of $\mathcal{P}$, for $i = 1, \ldots, m$:

$$\begin{aligned}\mathbb{E}[x_i(n+1)] \quad &= \quad p_i \cdot \mathbb{E}[a_i x_i(n) + P_i(x_1(n), \ldots, x_{i-1}(n))] \\ &\quad +(1-p_i) \cdot \mathbb{E}[b_i x_i(n) + Q_i(x_1(n), \ldots, x_{i-1}(n))].\end{aligned}$$

2: $MBRecs = \{\mathbb{E}[x_i(n+1)] \mid i = 1, \ldots, m\}$ ▷ initial set of moment-based recurrences
3: $S := \{x_1^k, \ldots, x_m^k\}$ ▷ initial set of monomials of E-variables as $Mom_k[0, x_i(n)] = \mathbb{E}[x_i(n)^k]$
4: **while** $S \neq \emptyset$ **do**
5: $M := \prod_{i=1}^{m} x_i^{\alpha_i} \in S$, where $\alpha_i \in \mathbb{N}$
6: $S := S \setminus \{M\}$
7: $M' = M[x_i^{\alpha_i} \leftarrow upd_i]$, for each $i = m, \ldots, 1$ ▷ replace each $x_i^{\alpha_i}$ in M with upd_i

where upd_i denotes:
$p_i \cdot (a_i x_i + P_i(x_1, \ldots x_{i-1}))^{\alpha_i} + (1-p_i) \cdot (b_i x_i + Q_i(x_1, \ldots x_{i-1}))^{\alpha_i}$

8: Rewrite M' as $M' = \sum N_j$ for monomials N_j over $x_1, \ldots, x_m$
9: Simplify the moment-based recurrence $\mathbb{E}[M(n+1)] = \mathbb{E}[\sum N_j]$ using the rules (5.4) ▷ $M(n+1)$ denotes $\prod_{i=1}^{m} x_i(n+1)^{\alpha_i}$
10: $MBRecs = MBRecs \cup \{\mathbb{E}[M(n+1)]\}$ ▷ add $\mathbb{E}[M(n+1)]$ to the set of moment-based recurrences
11: **for** each monomial N_j in M **do**
12: **if** $\mathbb{E}[N_j] \notin MBRecs$ **then** ▷ there is no moment-based recurrence for N_j
13: $S = S \cup \{N_j\}$ ▷ add N_j to S
14: **end if**
15: **end for**
16: **end while**
17: Solve the system of moment-based recurrences $MBRecs$
18: $MI = \{\mathbb{E}[x_i(n)^k] - CF_i(k,n) = 0 \mid i = 1, \ldots m\}$ ▷ $CF_i(k,n)$ is the closed form solution of $\mathbb{E}[x_i^k]$
19: **return** the set MI of moment based invariants of $\mathcal{P}$ for the kth moments of $x_1, \ldots, x_m$

Sketch. We want to show that $\mathbb{E}[x_i^{\alpha_i}]$ can be expressed using a recurrence equation. The idea is to express $x_i^{\alpha_i}(n+1)$ in terms of the value of x_i at nth iteration. The value of $x_i(n+1)$ is

$$a_i x_i(n) + P_i(x_1(n+1), \ldots x_{i-1}(n+1)) \tag{3.11}$$

with probability p and

$$b_i x_i(n) + Q_i(x_1(n+1), \ldots x_{i-1}(n+1)) \tag{3.12}$$

with probability $(1-p)$. From here, we can derive that

$$\mathbb{E}[x_i^{\alpha_i}(n+1)] = \mathbb{E}[p_i \cdot (a_i x_i + P_i(x_1, \dots x_{i-1}))^{\alpha_i} + (1-p_i) \cdot (b_i x_i + Q_i(x_1, \dots x_{i-1}))^{\alpha_i}]. \quad (3.13)$$

For arbitrary monomial $M = \prod x_i^{\alpha_i}(n+1)$, we can express $\mathbb{E}[M]$ by substituting each $x_i^{\alpha_i}(n+1)$ as above. This process is captured by line 7 of Algorithm 1. The new equations can be further simplified using properties of expected values and the simplification rules (5.4) to give recurrence equations over $\mathbb{E}$-variables. □

We now describe Algorithm 1. Our algorithm first rewrites $\mathcal{P}$ into a set $MBRecs$ of moment-based recurrences, as described in Section 3.2. That is, program variables x_i are turned into random variables $x_i(n)$ and variable updates over x_i become moment-based recurrences over $\mathbb{E}$-variables by using the relation of (3.4) (lines 1-2) of Algorithm 1).

The algorithm next proceeds with computing the moment-based recurrences of the kth moments of $x_1, \dots, x_m$. Recall that the kth moment of x_i is given by:

$$Mom_k[0, x_i(n)] = \mathbb{E}[x_i(n)^k].$$

Hence, the set S of monomials yielding $\mathbb{E}$-variables for which moment-based recurrences need to be solved is initialized to $\{x_1^k, \dots, x_m^k\}$ (line 3 of Algorithm 1). Note that by considering the resulting $\mathbb{E}$-variables $\mathbb{E}[x_i^k]$ and solving the moment-based recurrences of $\mathbb{E}[x_i^k]$, we derive closed forms of the kth moments of $\{x_1, \dots, x_m\}$ (line 17 of Algorithm 1). To this end, Algorithm 1 recursively computes the moment-based recurrences of every $\mathbb{E}$-variable arising from the moment-based recurrences of $\mathbb{E}[x_i^k]$ (lines 4-16 of Algorithm 1), thus ultimately computing closed forms for $\mathbb{E}[x_i^k]$. One can then use transformations described in Proposition 2.1 to compute closed forms for other moments, such as variance and covariance. In more detail,

- for each monomial $M = \prod x_j^{\alpha_j}$ from S, we substitute $x_i^{\alpha_i}$ in M by its probabilistic behaviour. That is, the update of x_i in the Prob-solvable loop $\mathcal{P}$ is rewritten, according to (3.4), into the sum of its two probabilistic updates, weighted by their respective probabilities (lines 5-7 of Algorithm 1). Rewriting in line 7 of Algorithm 1 represents the most non-trivial step in our algorithm, combining non-deterministic nature of our program with polynomial properties. The resulting polynomial M' from M is then reordered to be expressed as a sum of new monomials N_j (line 8 of Algorithm 1); such a sum always exists as M' involves only addition and multiplication over $x_1, \dots, x_m$ (recall that P_i and Q_i are polynomials over $x_1, \dots, x_m$).
- By applying the simplification rules(5.4) of $\mathbb{E}$-variables over the moment-based recurrence of $\mathbb{E}[\sum N_j]$, the recurrence of $\mathbb{E}[M(n+1)]$ is obtained and added to the set $MBRecs$. Here, $M(n+1)$ denotes $\prod_{i=1}^{m} x_i(n+1)^{\alpha_i}$. As the recurrence of $\mathbb{E}[M(n+1)]$ depends on $\mathbb{E}[N_j]$, moment-based recurrences of $\mathbb{E}[N_j]$ need also be computed and hence S is enlarged by N_j (lines 9-13 of Algorithm 1).

As a result, the set $MBRecs$ of moment-based recurrences over $\mathbb{E}$-variables corresponding to S is obtained. These recurrences are C-finite expressions over $\mathbb{E}$-variables (see correctness argument

of Theorem 3.3) and hence their closed-form solutions exist. In particular, the closed forms $CF_i(k,n)$ of $\mathbb{E}[x_i(n)^k]$ is derived, turning $\mathbb{E}[x_i(n)^k] - CF_i(k,n) = 0$ into a inductive property that holds at arbitrary loop iterations and is hence a moment-based invariant of $\mathcal{P}$ over the kth moment of x_i (line 17 of Algorithm 1).

Theorem 3.2 (Soundness)**.** *Consider a Prob-solvable loop $\mathcal{P}$ with program variables $x_1, \ldots, x_m$ and let k be a non-negative integer with $k \geq 1$. Algorithm 1 generates moment-based invariants of $\mathcal{P}$ over the kth moments of $x_1, \ldots, x_m$.*

Note when $k = 1$, Algorithm 1 computes the moment-based invariants as invariant relations over the closed-form solutions of expected values of $x_1, \ldots, x_m$. In this case, our moment-based invariants are quantitative invariants as in [KMMM10].

Example 3.3. *We illustrate Algorithm 1 for computing the second moments (i.e. $k = 2$) of the Prob-solvable loop of Figure 3.1(A). Our algorithm initializes with*

$$MBRecs = \{\mathbb{E}[f(n+1)], \mathbb{E}[x(n+1)], \mathbb{E}[y(n+1)], \mathbb{E}[s(n+1)]\}$$

and

$$S = \{f^2, x^2, y^2, s^2\}.$$

We next (arbitrarily) choose M to be the monomial f^2 from S. Thus, $S = \{x^2, y^2, s^2\}$. Using the probabilistic update of f, we replace f^2 by $\frac{3}{4} \cdot 1^2 + (1 - \frac{3}{4}) \cdot 0^2$, that is, by $\frac{3}{4}$. As a result, $MBRecs = MBRecs \cup \{\mathbb{E}[f(n+1)^2] = \frac{3}{4}\}$ and S remains unchanged.

We next choose M to be x^2 and set $S = \{y^2, s^2\}$. We replace x^2 by its randomised behaviour, yielding $\mathbb{E}[M(n+1)] = \mathbb{E}[x(n+1)^2] = \mathbb{E}[(x(n) + f(n+1) \cdot \texttt{ rand(1-d, 1+d)})^2]$. By the simplification rules (5.4) *over $\mathbb{E}$-variables, we obtain:*

$$\mathbb{E}[x(n+1)^2] = \mathbb{E}[x(n)^2] + 2 \cdot \mathbb{E}[x(n)] \cdot \mathbb{E}[f(n+1)] + \mathbb{E}[f(n+1)^2] \cdot \frac{1}{3}(d^2+3), \quad (3.14)$$

as $f(n+1)$ is independent from $x(n)$ and $\mathbb{E}[\texttt{ rand(1-d, 1+d)}^2] = \frac{1}{3}(d^2+3)$. We add the recurrence (3.14) *to $MBRecs$ and keep S unchanged as the $\mathbb{E}$-variables $\mathbb{E}[x(n)], \mathbb{E}[f(n+1)]$, and $\mathbb{E}[f(n+1)^2]$ have their recurrences already in $MBRecs$.*

We next set M to y^2 and change $S = \{s^2\}$. Similarly to $\mathbb{E}[x(n+1)^2]$, we get:

$$\mathbb{E}[y(n+1)^2] = \mathbb{E}[y(n)^2] + 4 \cdot \mathbb{E}[y(n)] \cdot \mathbb{E}[f(n+1)] + \mathbb{E}[f(n+1)^2] \cdot \frac{4}{3}(d^2+3), \quad (3.15)$$

by using that $f(n+1)$ is independent from $y(n)$ and $\mathbb{E}[\texttt{ rand(2-2d, 2+2d)}^2] = \frac{4}{3}(d^2+3)$. We add the recurrence (3.15) *to $MBRecs$ and keep S unchanged.*

We set M to s^2, yielding $S = \emptyset$. We extend $MBRecs$ with the recurrence:

$$\mathbb{E}[s(n+1)^2] = \mathbb{E}[(x(n+1) + y(n+1))^2] = \mathbb{E}[x(n+1)^2] + 2\mathbb{E}[(xy)(n+1)] + \mathbb{E}[y(n+1)^2]$$

and add xy to S. We therefore consider M to be xy and set $S = \emptyset$. We obtain:

$$\mathbb{E}[(xy)(n+1)] = \mathbb{E}[(xy)(n)] + 2 \cdot \mathbb{E}[x(n)] \cdot \mathbb{E}[f(n+1) + \mathbb{E}[y(n)] \cdot \mathbb{E}[f(n+1)] + 2 \cdot \mathbb{E}[f(n+1)^2],$$

by using that $\mathbb{E}[$`rand(1-d, 1+d)`$] = 1$ and $\mathbb{E}[$ `rand(2-2d, 2+2d)`$] = 2$. We add the recurrence of $\mathbb{E}[(xy)(n+1)]$ to $MBRecs$ and keep $S = \emptyset$.

As a result, we proceed to solve the moment-based recurrences of $MBRecs$. We focus first on the recurrences over expected values:

$$\begin{array}{lcll} \mathbb{E}[f(n+1)] & = & \frac{3}{4} & \\ \mathbb{E}[x(n+1)] & = & \mathbb{E}[x(n)] + \mathbb{E}[f(n+1) \cdot \texttt{rand(1-d, 1+d)}] & = \mathbb{E}[x(n)] + \frac{3}{4} \\ \mathbb{E}[y(n+1)] & = & \mathbb{E}[y(n)] + \mathbb{E}[f(n+1) \cdot \texttt{rand(2-2d, 2+2d)}] & = \mathbb{E}[x(n)] + 2 \cdot \frac{3}{4} \\ \mathbb{E}[s(n+1)] & = & \mathbb{E}[x(n+1)] + \mathbb{E}[y(n+1)] & \end{array}$$

Note that the above recurrences are C-finite recurrences over $\mathbb{E}$-variables. For computing closed forms, we respectively substitute $\mathbb{E}[f(n+1)$ by its closed form in $\mathbb{E}[y(n+1)]$ and $\mathbb{E}[x(n+1)]$, yielding closed forms for $\mathbb{E}[y(n+1)]$ and $\mathbb{E}[x(n+1)]$, and hence for $\mathbb{E}[s(n+1)]$. By also using the the initial values of Figure 3.1, we derive the closed forms:

$$\begin{array}{lcllcl} \mathbb{E}[f(n)] & = & \frac{3}{4} & \mathbb{E}[s(n)] & = & \frac{9}{4}n \\ \mathbb{E}[x(n)] & = & \frac{3}{4}n - 1 & \mathbb{E}[y(n)] & = & \frac{3}{2}n + 1 \end{array}$$

We next similarly derive the closed forms for higher-order and mixed moments:

$$\begin{array}{lcllcl} \mathbb{E}[f(n)^2] & = & \frac{3}{4} & \mathbb{E}[s(n)^2] & = & \frac{81}{16}n^2 + \frac{20d^2+27}{16}n \\ \mathbb{E}[x(n)^2] & = & \frac{9}{16}n^2 + \frac{4d^2-21}{16}n + 1 & \mathbb{E}[y(n)^2] & = & \frac{9}{4}n^2 + \frac{4d^2+15}{4}n + 1 \\ \mathbb{E}[(xy)(n)] & = & \frac{9}{8}n^2 - \frac{3}{8}n - 1 & & & \end{array}$$

yielding hence the moment-based invariants over the second moments of variables of Figure 3.1. Using Proposition 2.1 and Definition 2.8, we derive the variance $Var[s(n)] = \frac{20d^2+27}{16}n$.

Let us finally note that the termination of Algorithm 1 depends on whether for every monomial M (from the set S, line 4 of Algorithm 1) the moment-based recurrence equation over the corresponding $\mathbb{E}$-variable $\mathbb{E}[M(n+1)]$ can be computed as C-finite expression over $\mathbb{E}$-variables.

Theorem 3.3 (Termination)**.** *For any non-negative integer k with $k \geq 1$ and any Prob-solvable loop $\mathcal{P}$ with program variables $x_1, \ldots, x_m$, Algorithm 1 terminates. Moreover, Algorithm 1 terminates in at most $\mathcal{O}(k^m \cdot d_m^{m-1} \cdot d_{m-1}^{m-2} \ldots \cdot d_2^1)$ steps, where $d_i = max\{deg(P_i), deg(Q_i), 1\}$ with $deg(P_i), deg(Q_i)$ denoting the degree of polynomials P_i and Q_i of the variable updates* (3.2)*.*

Proof. We associate every monomial with an ordinal number as follows:

$$x_k^{\alpha_k} \cdot x_{k-1}^{\alpha_{k-1}} \ldots x_1^{\alpha_1} \xrightarrow{\sigma} \omega^k \cdot \alpha_k + \omega^{k-1} \cdot \alpha_{k-1} \cdots + \alpha_1,$$

and order monomials M, N such that $M > N$ if $\sigma(M) > \sigma(N)$. Algorithm 1 terminates if for every monomial M (from the set S, line 4 of Algorithm 1) the moment-based recurrence equation

over the corresponding $\mathbb{E}$-variable $\mathbb{E}[M(n+1)]$ can be computed as C-finite expression over $\mathbb{E}$-variables. We will show that this is indeed the case by transfinite induction over monomials.

Let $M = \prod_{k=1}^{K} x_k^{\alpha_k}$ be a monomial and assume that every smaller monomial has a closed form solution in form of a C-finite expression.

Let

$$x_i^{\alpha_i} := \left(c_i x_i + P_i(x_1, \cdots x_{i-1})\right)^{\alpha_i} \tag{3.16}$$

be the updates of our variables after removing the probabilistic choice as in line 5 of the algorithm. Then recurrence for M is

$$\begin{aligned}\mathbb{E}[M(n+1)] &= \mathbb{E}\Big[\prod_{i=1}^{K}\Big(p_i \cdot \left(a_i x_i + P_i(x_1, \dots x_{i-1})\right)^{\alpha_i} \\ &\qquad + (1-p_i)\cdot\left(b_i x_i + Q_i(x_1, \dots x_{i-1})\right)^{\alpha_i}\Big)(n)\Big] \\ &= \mathbb{E}[M(n)] + \sum_{j=1}^{J} b_j \cdot \mathbb{E}[N_j(n)] \end{aligned} \tag{3.17}$$

for some J, constants b_i and monomials $N_1, \dots, N_J$ all different than M. By Lemma 3.4, we have an inhomogeneous C-finite recurrence relation $\mathbb{E}[M(n+1)] = \mathbb{E}[M(n)] + \gamma$, for some C-finite expression γ. Hence, the closed form of $\mathbb{E}[M(n+1)]$ exists and is a C-finite expression. □

We finally prove our auxiliary lemma.

Lemma 3.4. *$M > N_j$ for all $j \leq J$ in* (3.17).

Proof. Let $M = \prod_{k=1}^{K} x_k^{\alpha_k}$ and have $N_j = \prod_{k=1}^{K} x_k^{\beta_k}$ coming from

$$\prod_{i=1}^{K} \left(c_i x_i + P_i(x_1, \cdots x_{i-1})\right)^{\alpha_i}. \tag{3.18}$$

Assume $M \leq N_j$, i.e. $\omega^K \cdot \alpha_K + \cdots + \alpha_1 \leq \omega^K \cdot \beta_K + \cdots + \beta_1$, so we have $\alpha_K \leq \beta_K$. Note that in (3.18) x_K only appears in factor $c_K x_K + P_K(x_1, \dots x_{K-1})$. Considering the multiplicity, we get at most α_Kth power of x_K, hence $\alpha_K \geq \beta_K$. Thus $\alpha_K = \beta_K$.

So for $M \leq N_j$ we need N_j from $(c_K x_K)^{\alpha_K} \cdot \prod_{i=1}^{K-1} \left(c_i x_i + P_i(x_1, \cdots x_{i-1})\right)^{\alpha_i}$.

Proceeding similarly for $x_{K-1}, x_{K-2}, \cdots$ we get that for each $k \leq K$ we have $\alpha_k = \beta_k$, which contradicts the assumption, thus $M > N_j$ as needed.

Regarding the termination, let's look at what monomials can possibly be added to S. Let $M = \prod x_i^{\alpha_i} \in S$. Based on the algorithm and the above, it is clear that in the case $i = m$ we have $\alpha_m \leq k$. For any $i < m$ the maximum value of α_i is $\alpha_{i+1} \cdot d_{i+1}$. Hence we have $\alpha_i \leq k \cdot \prod_{j=i+1}^{m} d_j$. thus we can count all possible monomials, hence the upper bound on the algorithm time complexity, as a product of these upper bounds. This yields $k^m \cdot d_m^{m-1} \cdot d_{m-1}^{m-2} \cdots \cdot d_2^1$ as claimed. □

3.4 Implementation and Experiments

We implemented our work in the Julia language, using Aligator ([HJK18a]) for handling and solving recurrences. We evaluated our work on several challenging probabilistic programs with parametrised distributions, symbolic probabilities and/or both discrete and continuous random variables. All our experiments were run on MacBook Pro 2017 with 2.3 GHz Intel Core i5 and 8GB RAM. Our implementation and benchmarks are available at: `github.com/miroslav21/aligator`.

Benchmarks We evaluated our work on 13 probabilistic programs, as follows. We used 7 programs from works [CHWZ15, KMMM10, CS14, FZJ+17, KUH19] on invariant generation. These examples are given in lines 1-7 of Table 3.1; we note though that BINOMIAL("p") represents our generalisation of a binomial distribution example taken from [CHWZ15, FZJ+17, KMMM10] to a probabilistic program with parametrised probability p. We further crafted six examples of our own, illustrating the distinctive features of our work. These examples are listed in lines 8-13 of Table 3.1: lines 8-11 correspond to the examples of Figure 3.1; line 12 of Table 3.1 shows a variation of Figure 3.1, with a parametrized distribution p; line 13 corresponds to a non-linear Prob-solvable loop computing squares. All our benchmarks are available at the aforementioned URL as well as at the end of this section.

Experimental results with moment-based invariants Results of our evaluation are presented in Table 3.1. While Algorithm 1 can compute invariants over arbitrary kth higher-order moments, due to lack of space and readability, Table 3.1 lists only our moment-based invariants up to the third moment (i.e. $k \leq 3$), that is for expected values, second- and third-order moments. The first column of Table 3.1 lists the benchmark name, whereas the second column gives the degree of the moments (i.e., $k = 1, 2, 3$) for which we compute invariants. The third column reports the timings (in seconds) our implementation needed to derive invariants. The last column shows our moment-based invariants; for readability, we decided to omit intermediary invariants (up to 30 for some programs) and only display the most relevant invariants.

We could not perform a fair practical comparison with other existing methods: to the best of our knowledge, existing works, such as [KMMM10, GKM13, BEFH16, KUH19], require user guidance/templates/hints. Further, existing techniques do not support symbolic probabilities and/or parametrised distributions - which are, for example, required in the analysis of programs STUTTERINGA, STUTTERINGC, STUTTERINGP of Table 3.1. We also note that examples COUPON, STUTTERINGC, STUTTERINGP involve non-linear probabilistic updates hindering automation in existing methods. In contrast, such updates can naturally be encoded as moment-based recurrences in our framework. We finally note that while second-order moments are computed only by [KUH19], but with the help of user-provided templates, no existing approaches compute moments for $k \geq 3$. Our experiments show that inferring third-order moments are in general, not expensive; yet, for examples STUTTERINGA, STUTTERINGC, STUTTERINGP with parametrized distributions/probabilities, more computation time is needed.

Program	Moment	Runtime (s)	Computed Moment-Based Invariants
COUPON [KUH19]	1	0.37	$\mathbb{E}[c(n)] = (2^n - 1)/(2^n)$
	2	0.40	$\mathbb{E}[c^2(n)] = (2^n - 1)/(2^n)$
	3	0.34	$\mathbb{E}[c^2(n)] = (2^n - 1)/(2^n)$
COUPON4 [KUH19]	1	0.90	$\mathbb{E}[c(n)] = (4^n - 3^3)/(4^n)$
	2	1.1	$\mathbb{E}[c^2(n)] = (4^n - 3^3)/(4^n)$
	3	1.3	$\mathbb{E}[c^3(n)] = (4^n - 3^3)/(4^n)$
RANDOM_WALK_1D_CTS [KUH19]	1	0.12	$\mathbb{E}[x(n)] = n/5$
	2	0.45	$\mathbb{E}[x^2(n)] = n^2/25 + 22n/75$
	3	1.00	$\mathbb{E}[x^3(n)] = n^3/125 + n^2 22/125 - n21/250$
SUM_RND_SERIES [CHWZ15]	1	0.31	$\mathbb{E}[x(n)] = n^2/4 + n/4$
	2	2.89	$\mathbb{E}[x^2(n)] = n^4/16 + 5n^3/24 + 3n^2/16 + n/24$
	3	17.7	$\mathbb{E}[x^3(n)] = n^6/64 + 7n^5/64 + 13n^4/64 + 9n^3/64 + n^2/32$
PRODUCT_DEP_VAR [CHWZ15]	1	0.65	$\mathbb{E}[p(n)] = n^2/4 - n/4$
	2	6.27	$\mathbb{E}[p^(n)] = n^4/16 - n^3/8 + 3n^2/16 - n/8$
	3	37.5	$\mathbb{E}[p^3(n)] = n^6/64 - 3n^5/64 + 9n^4/64 - 21n^3/64 + 15n^2/32 - n/4$
RANDOM_WALK_2D [CS14, KUH19]	1	0.07	$\mathbb{E}[x(n)] = 0$
	2	0.26	$\mathbb{E}[x^2(n)] = n/2$
	3	0.49	$\mathbb{E}[x^3(n)] = 0$
BINOMIAL("p") [CHWZ15, FZJ+17, KMMM10]	1	0.17	$\mathbb{E}[x(n)] = np$
	2	0.47	$\mathbb{E}[x^2(n)] = n^2p^2 + np(1-p)$
	3	1.6	$\mathbb{E}[x^3(n)] = n^3p^3 - 3n^2p^3 + 3n^2p^2 + 2np^3 - 3np^2 + np$
STUTTERINGA – FIG. 3.1(A)	1	0.44	$\mathbb{E}[s(n)] = 9n/4$
	2	2.2	$\mathbb{E}[s^2(n)] = 81n^2/16 + (20d^2 + 27)/16n$
	3	8.48	$\mathbb{E}[s^3(n)] = 81d^2n^2/16 + 63d^2n/16 + 729n^3/64 + 9n^2(4d^2 - 9)/32 + 9n^2(4d^2 + 9)/16 + 567n^2/64 + 3n(-6d^2 - 21)/8 + 3n(6d^2 - 12)/16 + 243n/32$
STUTTERINGB – FIG. 3.1(B)	1	0.49	$\mathbb{E}[s(n)] = 9n/4$
	2	2.03	$\mathbb{E}[s^2(n)] = 81n^2/16 + 347/16n + 128/3$
	3	7.43	$\mathbb{E}[s^3(n)] = 729n^3/64 + 9369n^2/64 + 1359n/32=$
STUTTERINGC – FIG. 3.1(C)	1	1.8	$\mathbb{E}[s(n)] = 3n^3/8 + 3n^2/8 - n$
	2	72.5	$\mathbb{E}[s^2(n)] = 9n^6/64 + 3n^5(8d^2 + 27)/160 + n^4(8d^4 + 84d^2 - 90)/192 + n^3(32d^4 + 216d^2 - 252)/288 + n^2(8d^4 + 44d^2 + 61)/64 + n(80d^4 + 324d^2 - 9)/1440$
	3	2144	$\mathbb{E}[s^3(n)] = 27n^9/512 + 27n^8(16d^2 + 39)/2560 + 3n^7(824d^4 + 6444d^2 + 1242)/17920 + n^6(1900d^4 + 3996d^2 - 4365)/2560 + n^5(2004d^4 + 1704d^2 - 54)/2560 + n^4(-1900d^4 - 7056d^2 + 13446)/7680 + n^3(-6948d^4 - 12708d^2 - 6969)/7680 + n^2(-1900d^4 - 3114d^2 - 315)/3840 + n(-108d^4 - 603d^2 + 288)/6720$
STUTTERINGD – FIG. 3.1(D)	1	1.92	$\mathbb{E}[s(n)] = 3n^3/8 + 3n^2/8 - n$
	2	46.3	$\mathbb{E}[s^2(n)] = 9n^6/64 + 93n^5/32 + 1651n^4/96 + 2849n^3/72 + 2813n^2/64 + 5131n/288$
	3	2076	$\mathbb{E}[s^3(n)] = 27n^9/512 + 1593n^8/512 + 94587n^7/1792 + 545971n^6/2560 + 270117n^5/1280 - 58585n^4/768 - 132599n^3/512 - 536539n^2/3840 - 771n/140$
STUTTERINGP	1	0.28	$\mathbb{E}[s(n)] = 3np$
	2	1.68	$\mathbb{E}[s^2(n)] = 11n^2p^2 + 3np(-2p+1) + np(-p-1) + 4np(-p+2) - 1$
	3	6.05	$\mathbb{E}[s^3(n)] = 27n_1^3p^3 - 3n_1^2p^3 + 3n_1^2p^2(-6p+3) + 12n_1^2p^2(-3p+3) + 12n_1^2p^2(-2p+3) + 3n_1p(4p^2 - 3p + 3) + 3n_1p(8p^2 - 12p + 9) + n_1p(p^2 - 3p(-p-1) - 3p + 2)/2 + 2n_1p(2p^2 - 6p(-p+2) - 6p + 13) + 6$
SQUARE	1	0.38	$\mathbb{E}[y(n)] = n^2 + n$
	2	2.46	$\mathbb{E}[y^2(n)] = n^4 + 6 * n^3 + 3 * n^2 - 2 * n$
	3	8.70	$\mathbb{E}[y^3(n)] = n^6 + 15 * n^5 + 45 * n^4 - 15 * n^3 - 30 * n^2 + 16 * n$

Table 3.1: Moment-based invariants of Prob-solvable loops, where n is the loop counter.

Benchmarks

We evaluated our work on 13 Prob-solvable programs. For each program, we computed moment-based invariants of the first, second, and third-order moment. The programs are listed below, with `u()` representing uniform distribution.

COUPON

Probabilistic model of Coupon Collector's program for two coupons, taken from [KUH19].

```
f := 0
c := 0
d := 0
while true:
    f := 1 [1/2] 0
    c := 1 - f + c*f
    d := d + f - d*f
```

COUPON4

Probabilistic model of Coupon Collector's program for four coupons, taken from [KUH19].

```
f := 0
g := 0
a := 0
b := 0
c := 0
d := 0
while true:
    f := 1 [1/2] 0
    g := 1 [1/2] 0
    a := a + (1-a)*f*g
    b := b + (1-b)*f*(1-g)
    c := c + (1-c)*(1-f)*g
    d := d + (1-d)*(1-f)*(1-g)
```

RANDOM_WALK_1D_CTS

A variation of random walk in one dimension with assignments from continuous distributions taken from [KUH19].

```
v := 0
x := 0
while true:
    v := u(0,1)
    x := x + v [7/10] x - v
```

SUM_RND_SERIES

A program modeling *Sum of Random Series* game taken from [CHWZ15].

```
n := 0
x := 0
while true:
    n := n + 1
    x := x + n [1/2] x
```

PRODUCT_DEP_VAR

A program modeling *Product of Dependent Random Variables* game taken from [CHWZ15].

```
f := 0
x := 0
y := 0
p := 0
while true:
    f := 0 [1/2] 1
    x := x + f
    y := y + 1 - f
    p := x*y
```

RANDOM_WALK_2D

A variation of random walk in two dimension as in [CS14, KUH19]. Each direction is chosen with equal probability.

```
h := 0
x := 0
y := 0
while true:
    h := 1 [1/2] 0
    x := x-h [1/2] x +h
    y := y+(1-h) [1/2] y-(1-h)
```

BINOMIAL

Another classic example, modeling binomial distribution. Appeared also in [CHWZ15, FZJ+17, KMMM10]. However, we consider the program to be parametric, computing invariants for arbitrary values of p.

```
x := 0
while true:
    x := x + 1 [p] x
```

STUTTERINGA

Program 3.1(A) from Section 3.1.

STUTTERINGB

Program 3.1(B) from Section 3.1.

STUTTERINGC

Program 3.1(C) from Section 3.1.

STUTTERINGD

Program 3.1(D) from Section 3.1.

STUTTERINGP

A variation of program 3.1(A) from Introduction with $d = 1$, parametrized w.r.t. p.

```
f := 0
x := -1
y := 1
s := 0
while true:
    f := 1 [p] 0
    x := x + f*u(0,2)
    y := y + f*u(0,4)
    s := x + y
```

SQUARE

Our own program with polynomial assignments.

```
x := 0; y := 1
while true:
    x := x+2 [1/2] x
    y := x^2
```

3.5 Chapter Conclusion

We introduced a novel approach for automatically generating *moment-based invariants* of a subclass of probabilistic programs (PPs), called Prob-solvable loops, with polynomial assignments over random variables and parametrised distributions. We combine methods from symbolic summation and statistics to derive invariants over higher-order moments, such as expected values or variances, of program variables. To the best of our knowledge, our approach is the first method for computing higher-order moments of PPs fully automatically and the first to handle PPs with parametrised distributions.

CHAPTER 4

Implementation - MORA

This chapter introduces tool MORA and is based on a joint work [BKS20b] with Laura Kovács and Ezio Bartocci, published in the proceedings of TACAS 2020.

[BKS20b] Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. Mora - automatic generation of moment-based invariants. In *Proc. of TACAS 2020: the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 12078 of *LNCS*, pages 492–498. Springer, 2020.

4.1 Overview

In this chapter, we introduce tool MORA. MORA implements the approach for automatically generating quantitative invariants of Prob-solvable loops, as described in Chapter 3, with random assignments, parametrized distributions, and probabilistic polynomial updates. The main purpose of this chapter is to describe what MORA can do and how it can be used. It is intended as a tool demonstration and a guide for the potential users. We focus on the usage and implementation aspects of MORA, with the details on theoretical foundations and algorithmic aspects of MORA being covered in Chapter 3. We note, however, that, when compared to the experimental setup from Chapter 3 and [BKS19], MORA comes with a completely new design, fully implemented in `python` and supporting easy installation and use even by non-experts in PP analysis. The implementation is available at:

`https://github.com/miroslav21/mora`,

and was successfully evaluated on a number of challenging examples. Unlike other existing approaches, e.g. [KMMM10, CS14, BEFH16, KUH19], MORA computes non-linear invariants in a fully automatic way without relying on user-provided templates/hints. The proposed automatic approach can handle an arbitrary number of loop iterations and infinite loops. On the contrary,

```
x=0
while true:
  u = RV(uniform, 0, b)
  g = RV(gauss, 0, 1)
  x = x - u @ 1/2; x + u @ 1/2
  y = y + x + g
```

Loop conditions are ignored, yielding non-deterministic PPs. The value of the random variable `u` is sampled from a uniform distribution with support in the real interval `[0, b]`, whereas the value of `g` is a random number from a Gaussian distribution with mean (first moment) 0 and variance (second moment) 1. Updates of program variable `x` are probabilistic: with probability `1/2`, the variable `x` is updated to `x-u`. Similarly, with probability `1/2`, `x` is updated to `x+u`. Further, updates to `u` and `g` do not depend on other variables; the update to `x` depends only on itself and `u`.

Figure 4.1: An illustrative example of a Prob-solvable loop.

tools like PSI [GMV16] support only the automatic analysis of probabilistic programs with a specified number of loop iterations.

Moreover, the invariants inferred by MORA are not restricted to expected values but are quantitative invariants over the higher-order moments of program variables. We refer to such invariants as *moment-based invariants* [BKS19]. To the best of our knowledge, no other tool can automatically compute higher-order moments of PPs, not even for the restricted, yet expressive enough, class of Prob-solvable loop.

4.2 MORA – Programming Model

Input programs to MORA are PP loops that are Prob-solvable [BKS19]. In Figure 4.1, we give an example of a Prob-solvable loop and use this example as a running example to guide the potential users of MORA in the rest of this chapter.

In a nutshell, probabilistic assignments of Prob-solvable loops include (i) variable values drawn from random distributions, such as uniform or normal distributions, and (ii) random variable updates. In the sequel, we write `RV` to refer to a random variable. Input programs to MORA therefore satisfy the following two properties:

(1) Input programs to MORA are PPs generated from the grammar in Figure 4.2.

(2) In addition to the grammar of Figure 4.2, MORA requires its PP input to be Prob-solvable, imposing further restrictions as follows:

- PP loop variables are different from each other and from parameters;
- probabilities used within a variable update sum up to 1;
- updated variables depend on themselves linearly and may depend polynomially only on other variables that have been previously updated.

Grammar defining PP inputs to MORA

```
PROGRAM → INIT_ASSIGNS "while true:" RV_ASSIGNS UPD_ASSIGNS

INIT_ASSIGNS → INIT_ASSIGN | INIT_ASSIGN INIT_ASSIGNS
RV_ASSIGNS → RV_ASSIGN | RV_ASSIGN RV_ASSIGNS
UPD_ASSIGNS → UPD_ASSIGN | UPD_ASSIGN UPD_ASSIGNS

INIT_ASSIGN → VAR " = " INIT_EXPR
RV_ASSIGN → VAR " = " RV_EXPR
UPD_ASSIGN → VAR " = " UPD_BRANCHES

UPD_BRANCHES → UPD_BRANCH | UPD_BRANCH UPD_BRANCHES
UPD_BRANCH → UPD_EXPR "@" UPD_PROB
UPD_PROB → SIMP_EXPR

INIT_EXPR → RV_EXPR | SIMP_EXPR
RV_EXPR → "RV(uniform, " SIMP_EXPR ", " SIMP_EXPR ")"
        | "RV(gauss, " SIMP_EXPR ", " SIMP_EXPR ")"
UPD_EXPR → UPD_EXPR OP UPD_EXPR | VAR | ATOM
SIMP_EXPR → SIMP_EXPR OP SIMP_EXPR | ATOM

ATOM → NUM | PARAMETER
OP → [*+-]
VAR → [a-zA-Z][a-zA-Z0-9]*
PARAMETER → [a-zA-Z][a-zA-Z0-9]*
NUM → [-]?[0-9]+[.]?[0-9]*([\/][1-9][0-9]*)?
```

Figure 4.2: Grammar of Prob-solvable loops for MORA

Note that Figure 4.1 satisfies all constraints above, and thus is Prob-solvable.

4.3 MORA– Usage

We describe the easiest way MORA can be used to generate moment-based invariants:

1. Save a Prob-solvable loop to a file, for example save Figure 4.1 in the file `running`.
2. In the main MORA folder invoke `python` with `python3.7` and execute:

```
from mora.mora import mora
```

3. Run MORA using the command:

```
mora("running", goal=GOAL),
```

where `GOAL` can be (i) a specific natural number $k \geq 1$, in which case MORA computes the kth moments of all variables from `running`; (ii) a specific moment of one loop variable of

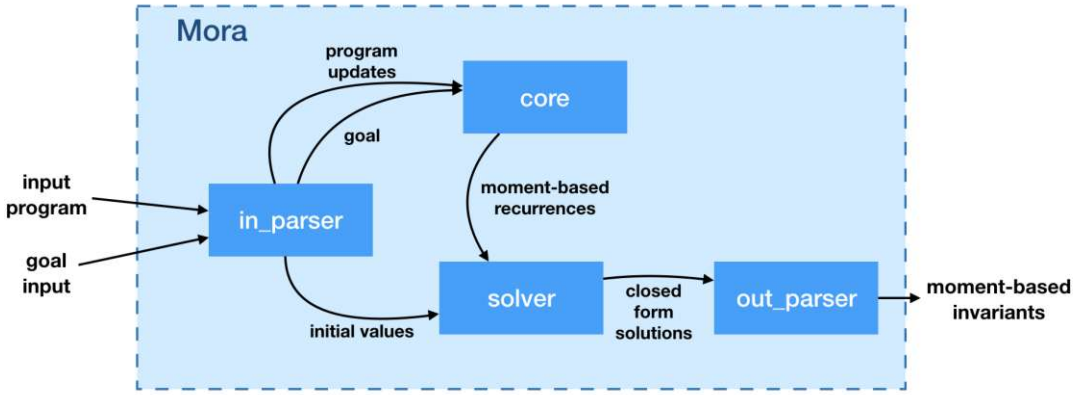

Figure 4.3: MORA workflow diagram.

`running` (e.g. `"x^2"` specifying the second moment of a variable `x` of Figure 4.1); or (iii) a list containing the goals as just specified. One can specify finitely many goals as inputs to MORA; yet, at least one goal is required. For example, by running `mora("running", [1, "x^2", "x^3"])`, MORA computes the expected values (first moments) of all variables from Figure 4.1, as well as the second and third moments of variable `x` of Figure 4.1 (specified by `x^2` and `x^3`, respectively).

MORA is completely automatic. That is, once execution of MORA is started on a given Prob-solvable loop and input goals, MORA outputs the higher-order moments, and thus moment-based invariants, of its loop w.r.t. the specified input goals. To this end, MORA computes the expected values of all monomials over loop variables, on which one of the goals from `GOAL` depends. In general, computing the `k`th moment requires computing the expected values of all monomial expressions over loop variables, such that the total degree of the monomials is less or equal than k, see [BKS19] for more details.

In the rest of the chapter, we will illustrate the main steps of MORA, by considering Figure 4.1 as its input loop and `[1, 2]` as its list of input goals. With such an input goal, MORA is set to compute the first and second moments of each variable of Figure 4.1. Note, that even if `1` was omitted from the aforementioned input goal, MORA would still need to compute some of the first moments of the variables, as they are required for computing the second-order moments. In the following, we showcase the MORA behaviour for:

```
mora("running", [1, 2]).
```  (4.1)

4.4 MORA– Tool Overview

We first give details on our implementation. We then present the overall workflow of MORA in Figure 4.3, based on which we overview the main components of our tool.

Overall implementation MORA is implemented in `python3`, requiring `python` version of at least 3.7. MORA relies on the `diofant` and `scipy` libraries: (i) the `python` library `diofant` is used in MORA for symbolic mathematical computations and recurrence solving; (ii) the `scipy` library, and in particular its statistics module `scipy.stats`, is used in MORA to handle probability distributions and statistical functions, as well as to simplify and compute expressions involving probability distributions and initial values of variables. Altogether, our implementation comprises around 350 lines of code.

MORA – Parser MORA first checks whether a given input program is Prob-solvable, by checking the requirements of Section 4.2. If the input program is not Prob-solvable, an error is reported, and the execution of MORA stops. Otherwise, within its parser module, MORA extracts initial values from its input loop, rewrites loop updates into equations over expected values of monomial expressions over loop variables, and processes the list of its input goals to identify which higher-order moments need to be computed.

For our demo execution (4.1), MORA extracts the initial value `x(0)=0`, where `x(0)` denotes the initial value of `x` before the loop. Using the input goals specified in (4.1), MORA is set to compute the expected values of `{u, g, x, y, u^2, g^2, x^2, y^2` characterizing the first and second moments of all loop variables of Figure 4.1. Further, the loop updates of Figure 4.1 are rewritten by MORA into equations over expected values, as follows:

$$\begin{cases} \mathbb{E}[x^k(n+1)] = \mathbb{E}[1/2 \cdot (x(n) - u(n+1))^k + 1/2 \cdot (x(n) + u(n+1))^k] \\ \mathbb{E}[y^k(n+1)] = \mathbb{E}[(y(n) + x(n+1) + g(n+1))^k] \end{cases}, \tag{4.2}$$

where $n \geq 0$ is the loop counter of Figure 4.1, $x(n)$ denotes the value of `x` at the nth loop iteration, and $\mathbb{E}[expr]$ is the expected value of an expression $expr$.

MORA – Core After rewriting probabilistic loop updates into equations over expected values, MORA rewrites these equations into non-probabilistic recurrences over so-called $\mathbb{E}$-variables, with the loop counter n being the recurrence index. $\mathbb{E}$-variables are simply variables created from monomials over original variables. Thanks to the restrictions defining PPs to be Prob-solvable, the resulting recurrences are linear recurrences with constant coefficients, that is, C-finite recurrences, whose closed forms can always be computed [KP11]. MORA solves these recurrences by calling its *Solver* module.

Using the equations (4.2) over expected values, the non-probabilistic recurrences of Figure 4.1 generated by MORA are as follows, using the MORA synthax:

```
y = x + y
g**2 = 1
x = x
u = b/2
x**2 = b**2/3 + x**2                                    (4.3)
u**2 = b**2/3
y**2 = b**2/3 + x**2 + 2*x*y + y**2 + 1
g = 0
x*y = b**2/3 + x**2 + x*y
```

The left-hand sides of these equations represent values of $\mathbb{E}$-variables at iteration $n+1$, while monomials over original variables on the right-hand side represent $\mathbb{E}$-variables at iteration n. For example, the first equation of (4.3) stands for $\mathbb{E}[y(n+1)] = \mathbb{E}[x(n)] + \mathbb{E}[y(n)]$. On the other hand, the fourth equation of (4.3) represents $\mathbb{E}[x(n+1)^2] = \frac{b^2}{3} + \mathbb{E}[x(n)^2]$, as b is a constant parameter and `x**k` in `python` denotes the kth power of x.

Solver In this module, MORA extracts and solves recurrences from the non-probabilistic equations over $\mathbb{E}$-variables computed by its *Core* module. By exploiting the structure of Prob-solvable programs, MORA also optimizes the order in which recurrences are solved, e.g. independent recurrences are solved first. Partial solutions can be used to reduce the complexity of the latter recurrences. MORA then uses the `diofant` library to handle and solve single recurrences.

For Figure 4.1, using the $\mathbb{E}$-variable equations of (4.3), the following closed form solutions are computed by MORA:

$$\begin{aligned}
&\mathbb{E}[u^2] = \tfrac{b^2}{3}\\
&\mathbb{E}[x^1] = 0\\
&\mathbb{E}[y^1] = y(0)\\
&\mathbb{E}[x^2] = \tfrac{b^2 n}{3}\\
&\mathbb{E}[u^1] = \tfrac{b}{2}\\
&\mathbb{E}[y^1 x^1] = \tfrac{b^2 n}{6}(n+1)\\
&\mathbb{E}[y^2] = \tfrac{n}{18}\left(2b^2n^2 + 3b^2n + b^2 + 18\right) + y(0)^2\\
&\mathbb{E}[g^1] = 0\\
&\mathbb{E}[g^2] = 1
\end{aligned} \tag{4.4}$$

with $y(0)$ standing for the initial value of y (treated as a parameter, since not specified).

MORA – Out_Parser MORA's output consists of basic information about the program and the goal, moment-based invariants computed, and computation time. By default, the MORA output is shown only on the screen. However, an optional argument can specify if an output file should be created. Two possible values for `output_format` are (i) `"txt"`, producing a simple human-readable file, and (ii) `"tex"`, producing a file with invariants in LaTeX format (as given in (4.4) above).

Program	Moment	Runtime PoC (s)	Runtime MORA (s)
SUM_RND_SERIES	1	0.31	0.22
	2	2.89	0.93
	3	17.7	2.47
STUTTERINGA	1	0.44	0.25
	2	2.20	1.07
	3	8.48	3.35
STUTTERINGC	1	1.80	0.66
	2	72.5	12.2
	3	2144	73.9
SQUARE	1	0.38	0.22
	2	2.46	0.73
	3	8.70	1.67

Table 4.1: Comparison of MORA vs. proof-of-concept (PoC) implementation of [BKS19].

4.5 Evaluation

A proof-of-concept implementation, together with initial experiments, were already given in our work on generating moment-based invariants [BKS19]. MORA comes, however, with a new design and a re-implementation of [BKS19], significantly improving the experimental setting and evaluations of [BKS19]. Table 4.1 compares MORA against the experiments of [BKS19], on a subset of Prob-solvable loops from [BKS19], evidencing that MORA is faster than our initial proof-of-concept implementation. This is due to the following reasons:

- MORA now optimizes the order in which recurrences are sent to the `diofant` recurrence solver. This reduces the amount of necessary symbolic computation and speeds up the process.
- While MORA is implemented entirely in `python`, with limited usage of external libraries, the previous implementation was done in `Julia` and relied on calls to the `sympy` library of `python`.
- MORA does not rely on `Aligator` [HJK18a] for handling systems of recurrences, allowing us to eliminate some intermediate and redundant steps.

4.6 Chapter Conclusion

We described MORA, a fully automated tool for generating invariants of probabilistic programs. MORA combines recurrence solving, symbolic summation, and statistical reasoning and derives higher-order moments of loop variables in probabilistic programs.

CHAPTER 5

Analyzing Bayesian Networks

This chapter is based on a joint work [BKS20a, SBK22] with Laura Kovács and Ezio Bartocci, published in the proceedings of ICTAC 2020, and in the Theoretical Computer Science journal, respectively.

[BKS20a] Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. Analysis of Bayesian networks via prob-solvable loops. In *Proc. of ICTAC 2020: the 17th International Colloquium on Theoretical Aspects of Computing*, volume 12545 of *LNCS*, pages 221–241. Springer, 2020.

[SBK22] Miroslav Stankovič, Ezio Bartocci, and Laura Kovács. Moment-based analysis of bayesian network properties. *Theoretical Computer Science*, 903:113–133, 2022.

5.1 Overview

Bayesian networks, or BNs for short, are well-established probabilistic models suitable to represent, learn and predict (through Bayesian inference) the behavior of complex systems with uncertainty or partially available information. The emergence of BNs, starting with the seminal work [Pea85], represented an important milestone in the advancement of artificial intelligence. The scientific impact of these models is also confirmed by the multitude of fields where they found a successful application. Examples include probabilistic machine learning [Hec08], speech recognition [ZR98], sports betting [CFN12], image processing [LSS05], runtime verification [KBS+13], program synthesis [SRB+15], gene regulatory networks [FLNP00], diagnosis of diseases [JC10], and finance [NJ07].

A BN is a graphical representation of a probability distribution over a set of random variables. In particular, it consists of a directed acyclic graph (DAG), where the nodes represent random variables and edges capture their conditional dependencies. Each node/variable is associated with

a probability distribution that is conditional on each value combination of the nodes/variables that are its parents in the DAG.

For example, Figure 5.1 shows a BN modeling two events, the rain (R) and an active sprinkler (S) that can be responsible for the grass (G) being wet. As expected, the sprinkler is usually inactive when it rains. This means that the probability of the sprinkler being active depends on the probability to rain. The BN in Figure 5.1 models graphically such dependency with a directed edge starting from the node representing the variable R and ending to the node associated with the variable S. This dependency is also captured quantitatively by a *conditional probability table* (CPT) associated with the variable S. A CPT specifies, for each possible combination of values of the parents' variables (one for each row of the table), and the corresponding probability for the child's variable to have a certain discrete value (one for each table column). In our running example of Figure 5.1, G, R, S are binary random variables with Bernoulli conditional distributions. However, in general, BNs allow arbitrary types for their random variables and their conditional distributions.

Probabilistic inference A common operation on BNs is to estimate what is the probability of an event being the cause of another observed event. For example, in the BN of Figure 5.1, we may want to answer the following question:

Q1 - What is the probability that it has rained, given that the grass is wet?

The inherited Bayesian inference framework of BNs provides a solution to this problem, enabling the computation of the posterior probability of some random variables, given the prior distribution of other random variables in the network. The problem of probabilistic inference has been extensively investigated in the literature [KF09] resulting in the development of *exact* and *approximation* (using Monte Carlo simulations) techniques, both found to be computationally NP-hard [Coo90, DL93].

Number of samples Among the approximation techniques [KF09, YD06], a typical example is the *rejection sampling*: a sample is accepted when it complies with the evidence; otherwise is rejected. A drawback of this method is the number of samples that it may require before getting the first accepted sample, while most of the samples may be wasted simply because they do not satisfy the observations. Thus, a second important question, investigated also in [BKKM18], is:

Q2 - What is the expected number of samples until an accepting sample occurs?

Sensitivity analysis BN parameters are most likely to be imprecise or wrong because they are often provided manually or estimated from (incomplete) data. In the example of Figure 5.1, the CPT for the random variable S contains imprecise symbolic parameters a and b. In this case, *sensitivity analysis* aims to answer this third question:

Q3 - How much does a small change in BN parameters affect probabilistic inference?

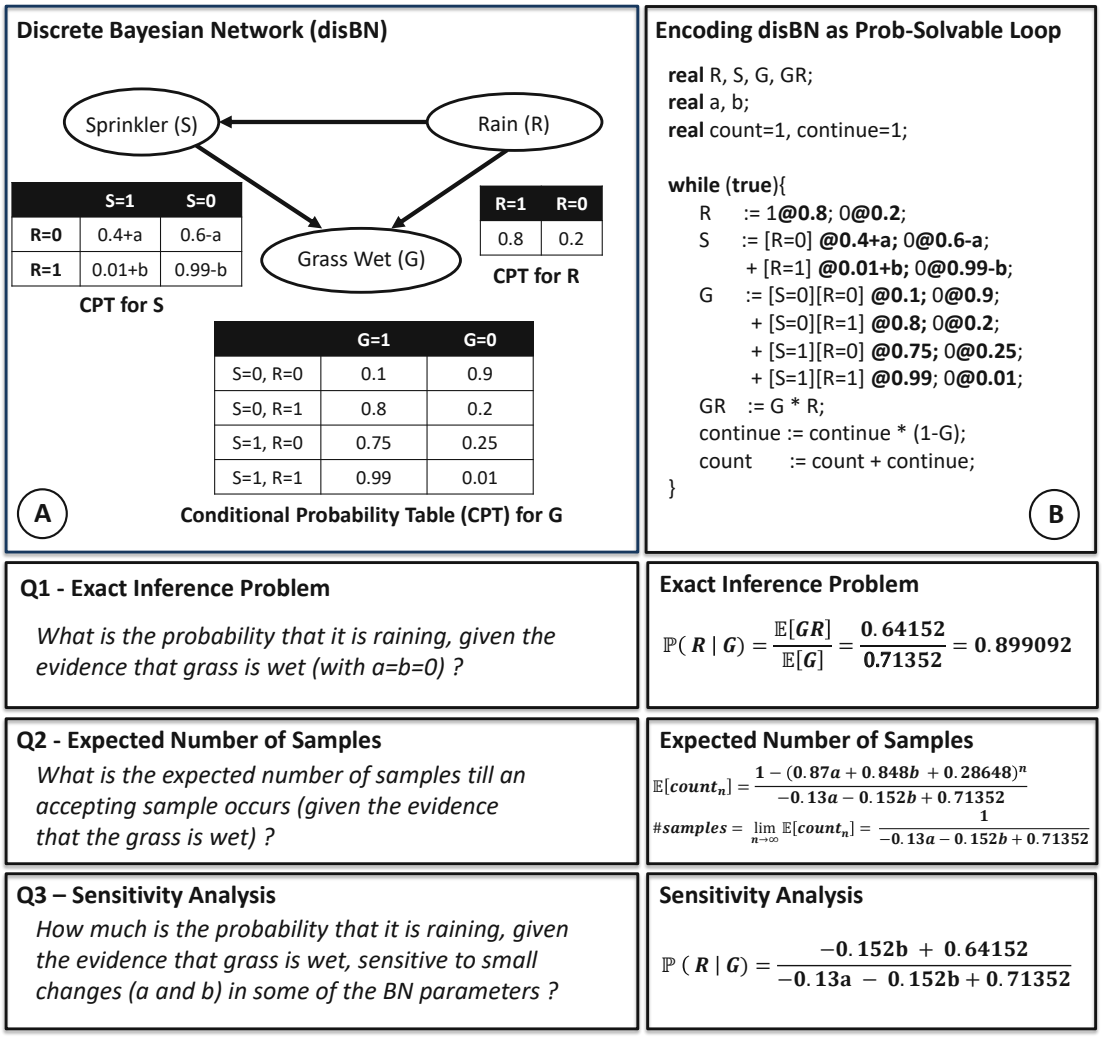

Figure 5.1: Solving probabilistic inference, the expected number of samples, and the sensitivity analysis for a discrete BN (disBN), by encoding the disBN as a Prob-solvable loop and computing automatically moment-based invariants (MBIs).

Analysis of BNs as PPs Probabilistic programs provide a unifying framework to both encode probabilistic graphical models, such as BNs, and to implement sophisticated inference algorithms and decision-making routines that can operate in real-world applications [Gha15]. In this chapter, we explore the link between Bayesian network analysis and static program analysis. Specifically, we connect BN analysis with the moment-based analysis of probabilistic programs. An extension of Prob-solvable loops is proposed, with new features essential for encoding BNs and for solving several kinds of BN analyses via invariant generation over higher-order statistical moments of Prob-solvable loop variables. Figure 5.1(B) shows a Prob-solvable loop encoding the probabilistic behavior of the discrete BN (disBN) illustrated in Figure 5.1(A). The Prob-solvable loop of Figure 5.1(B) requires one variable for each disBN node and some extra variables that depend on the particular BN analysis. For example, to solve exact probabilistic inference and sensitivity

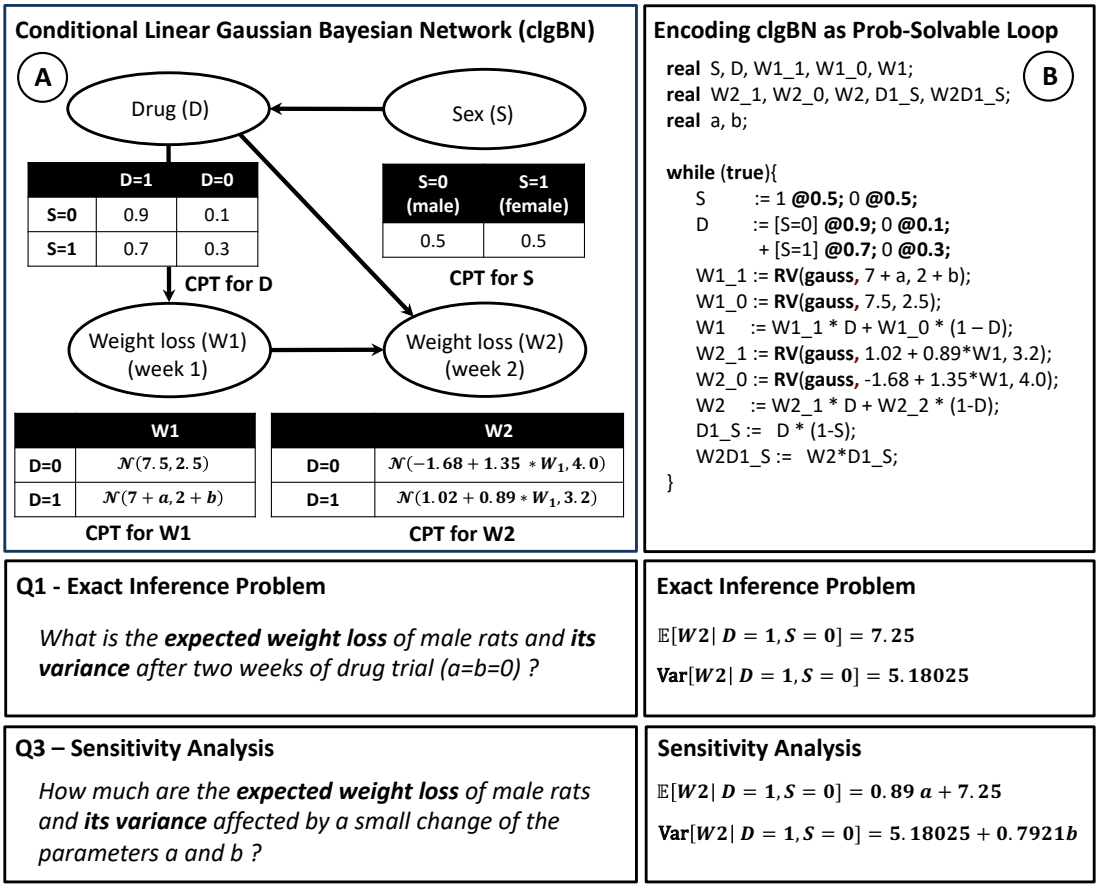

Figure 5.2: Solving probabilistic inference and sensitivity analysis in a conditional linear Gaussian BN (clgBN), by encoding the clgBN as a Prob-solvable loop and computing MBIs.

analysis, we require an extra variable to store the product of the random variables G and R. On the other hand, to compute the expected number of samples until an accepting sample occurs, we would need two other auxiliary variables $count$ and $continue$. Each row of each CPT is encoded as a probabilistic assignment in the Prob-solvable loop. Our approach generates moment-based invariants as quantitative invariants over higher-order moments to solve the three questions (Q1-Q3) of Figure 5.1. The required Prob-solvable loop analysis, however, requires additional steps (e.g. calculating limits) that were not yet supported by the Prob-solvable model [BKS19] of Chapter 3. Moreover, while the Prob-solvable programming model can model the probabilistic behavior of disBNs, it cannot model other BN variants, such as BNs with Gaussian conditional dependencies as in Figure 5.2(A). In this chapter, we also extend Prob-solvable loops with new features supporting Gaussian and uniform random variables depending on other random variables and show that these extensions allow us to solve BN problems via Prob-solvable loop reasoning.

Chapter contributions Let us briefly summarize the contributions presented in this chapter. (1) We extend the model of Prob-solvable loops and prove that it admits a decision procedure for computing moment-based invariants (Section 5.2). (2) We provide a sound encoding of BNs as Prob-solvable loops, in particular addressing discrete BNs (disBNs), Gaussian BNs (gBNs), conditional linear Gaussian BNs (clgBNs) and dynamic BN (dynBNs) (Section 5.3). (3) We formalize several BN problems as moment-based invariant generation tasks in Prob-solvable loops (Section 5.4). (4) We extend the tool MORA and evaluate our approach on a number of examples, fully automating BN analysis via Prob-solvable loop reasoning (Section 5.5).

The rest of this chapter is organized as follows. In Section 5.2, we introduce our probabilistic programming model while Section 5.3 and Section 5.4 show how to encode, respectively, both Bayesian Networks and different kinds of analyses in our framework. Section 5.5 provides the empirical evaluation of our approach applied to a set of benchmarks. We summarize this Chapter in Section 5.6.

5.2 Programming Model: Extending Prob-solvable Loops

We introduce our programming model extending the class of *Prob-solvable loops* [BKS19], allowing us to encode and analyze BN properties in Section 5.3. In particular, we extend [BKS20a] to allow coefficients to be polynomials over finite discrete variables and to support more complex expressions, which helps reduce the number of auxiliary variables necessary to encode BNs. To this end, we consider probabilistic while-programs as introduced in [Koz81, MM05] and restrict this class of programs to probabilistic programs with polynomial updates among random variables. In the following sections, whenever we refer to a Prob-solvable loop/program, we mean a program as defined below.

Definition 5.1 (Prob-solvable loop)**.** *Let $m, r \in \mathbb{N}_0$ and let $\mathcal{V}_D = \{d_i \mid i \leq r\}$ be the set of bounded discrete program variables and $\mathcal{V}_X = \{x_i \mid i \leq m\}$ be the set of arbitrary program variables. Let further $\mathcal{V} = \mathcal{V}_D \cup \mathcal{V}_X$. A* Prob-solvable loop *with variables from $\mathcal{V}$ is a probabilistic program of the form*

$$I; \texttt{while(true)}\{U\}, \tag{5.1}$$

where:

- *(Initialization) I is a sequence of initial assignments over $\mathcal{V}$. That is, I is an assignments sequence $v := c_v$, where c_v represents a number drawn from a known distribution* [1]*. In particular, c_v can be a constant. For $v \in \mathcal{V}_D$ we further require the distribution to be discrete and bounded.*
- *(Update) U denotes a sequence of $r+m$ random updates in the order $d_1, \ldots, d_r, x_1, \ldots, x_m$, with each update of the form:*

$$v := \sum_j upd_{v,j}$$

[1] a known distribution is a distribution with known and computable moments

where the $upd_{v,j}$ *has the form*

$$P_{i,j,1}(d_1,\ldots,d_{i-1})@p_{i,j,1};\ \ldots;\ P_{i,j,\alpha_{j,i}}(d_1,\ldots,d_{i-1})@p_{i,j,\alpha_{j,i}}; \tag{5.2}$$

if $v = d_i \in \mathcal{V}_D$ *or*

$$\begin{aligned} &Q_{i,j,1}(d_1,\ldots,d_r)x_i + R_{i,j,1}(d_1,\ldots,d_r,x_1,\ldots,x_{i-1})@q_{i,j,1};\\ &\quad\vdots\\ &Q_{i,j,\beta_{j,i}}(d_1,\ldots,d_r)x_i + R_{i,j,\beta_{j,i}}(d_1,\ldots,d_r,x_1,\ldots,x_{i-1})@q_{i,j,\beta_{j,i}}; \end{aligned} \tag{5.3}$$

if $v = x_i \in \mathcal{V}_X$*, where all* P*,* Q *and* R *are polynomials, and all* p *and* q *are probabilities of taking the respective value. An expression* $upd@p$*; means that the value* upd *is taken with probability* p*.*

- *(Dependencies) The coefficients of* Q *and* R *in the variable assignments* (5.2)-(5.3) *of* x_i *can be drawn from a random distribution as long as the moments of this distribution are known, and either they are (i) Gaussian or uniform distributions linearly depending on variables* x_j *with* $j \leq i$ *and polynomially on variables from* $\mathcal{V}_D$*; or (ii) other known distributions independent from all* $v \in \mathcal{V}$*.*

Note that Prob-solvable loop supports parametrised distributions. For example, one may have the uniform distribution $\mathcal{U}(l,u)$ with arbitrary symbolic constants $l < u \in \mathbb{R}$. Similarly, the probabilities p_i, q_i in the probabilistic updates (5.2)-(5.3) can be symbolic constants.

The restriction on random variable dependencies from Definition 5.1 extends [BKS19] by allowing parameters of Gaussian and uniform distributions in Prob-solvable loop to be specified using the previously updated program variables x_j and to depend on x_i linearly. In Theorem 5.1 we prove that this extension maintains the existence and computability of higher-order statistical moments of Prob-solvable loops, allowing us to derive all *moment-based invariants* of Prob-solvable loops of degree $k \geq 1$.

We also further allow the coefficients of (5.2)-(5.3) to be polynomials over discrete variables $\mathcal{V}_D$. We allow the updates to capture multiple non-binary probabilistic branches in the update, which further generalizes [BKS19, BKS20a] where updates were restricted to the form

$$x_i := a_i x_i + P_i(x_i,\ldots,P_{i-1})@p_i; b_i x_i + Q_i(x_i,\ldots,P_{i-1})@1-p_i;.$$

In the sequel, we prove that arbitrary moment-based invariants for our generalized class of Prob-solvable loops are computable (Theorem 5.2) using Algorithm 2.

Definition 5.2 (Moment-based invariants (MBIs))**.** *Let* $\mathcal{P}$ *be a Prob-solvable loop and* $n \in \mathbb{N}$ *denote an arbitrary loop iteration of* $\mathcal{P}$*. Consider* $k \in \mathbb{N}$ *with* $k \neq 0$*. A* moment-based invariant (MBI) of degree k over x_i *of* $\mathcal{P}$ *is* $\mathbb{E}[x_i(n)^k] = f_{x_i,k}(n)$*, where* $f_{x_i,k} : \mathbb{N} \to \mathbb{R}$ *of* n *is a closed-form expression denoting the* k*th (raw) higher-order moments of* x_i*, such that* $f_{x_i,k}(b)$ *depends only* n *and the initial variable values of* $\mathcal{P}$*.*

Algorithm 2 Moment-Based Invariants (MBIs) of generalized Prob-solvable Loops

Input: Prob-solvable loop $\mathcal{P}$ with variables $\mathcal{V} = \{x_1, \ldots, x_m\}$, and $k \geq 1$
Output: MBIs of $\mathcal{P}$ of degree k
Assumptions: $n \in \mathbb{N}$ is an arbitrary loop iteration of $\mathcal{P}$

1: Extract moment-based recurrence relations of $\mathcal{P}$, for $v \in \mathcal{V}$:

$$\mathbb{E}[v(n+1)] = \sum_j upd_{v,j}$$

▷ as of Definition 5.1
2: $MBRecs = \{\mathbb{E}[v(n+1)] \mid v \in \mathcal{V}\}$ ▷ initial set of moment-based recurrences
3: $S := \{v^k \mid v \in \mathcal{V}\}$ ▷ initial set of monomials of $\mathbb{E}$-variables
4: **while** $S \neq \emptyset$ **do**
5: $M := \prod_{i=1}^{m} x_i^{\alpha_i} \in S$, where $\alpha_i \in \mathbb{N}$
6: $S := S \setminus \{M\}$
7: $M' = M[x_i^{\alpha_i} \leftarrow \sum_j upd_{x_i,j}]$, for each $i = m, \ldots, 1$ ▷ replace each $x_i^{\alpha_i}$ in M with $upd_{x_i,j}$ as in (5.7)
8: Rewrite M' as $M' = \sum N_j$ for monomials N_j over $x_1, \ldots, x_m$
9: **Simplify moment-based recurrence** $\mathbb{E}[M(n+1)] = \mathbb{E}[\sum N_j]$ **using** (5.4)-(5.6)
▷ $M(n+1)$ denotes $\prod_{i=1}^{m} x_i(n+1)^{\alpha_i}$
10: $MBRecs = MBRecs \cup \{\mathbb{E}[M(n+1)]\}$ ▷ add $\mathbb{E}[M(n+1)]$ to the set of moment-based recurrences
11: **for** each monomial N_j in M **do**
12: **if** $\mathbb{E}[N_j] \notin MBRecs$ **then** ▷ no moment-based recurrence for N_j
13: $S = S \cup \{N_j\}$ ▷ add N_j to S
14: **end if**
15: **end for**
16: **end while**
17: $MBI = \{\mathbb{E}[v(n)^k] - f_{v,k}(n) = 0 \mid v \in \mathcal{V}\}$ ▷ $f_{v,k}(n)$ is the closed form solution of $\mathbb{E}[x_i^k]$
18: **return** MBIs of $\mathcal{P}$ for the kth moments of $x_1, \ldots, x_m$

In what follows, we consider an arbitrary Prob-solvable loop $\mathcal{P}$ and formalize our results relative to $\mathcal{P}$. Further, we reserve $n \in \mathbb{N}$ to denote an arbitrary loop iteration of $\mathcal{P}$. Note that MBIs of $\mathcal{P}$ yield functional representations of the kth higher-order moments of loop variables x_i at n. Hence, the MBIs $\mathbb{E}[x_i(n)^k] = f_{x_i,k}(n)$ are valid and invariant. In Algorithm 2 we show that MBIs of Prob-solvable loops can always be computed. As in [BKS19], the main ingredient of Algorithm 2 are so-called $\mathbb{E}$-*variables* for capturing expected values and other higher-order moments of loop variables of $\mathcal{P}$.

Definition 5.3 ($\mathbb{E}$-variables of Prob-solvable Loops [BKS19])**.** *An* $\mathbb{E}$-variable *of* $\mathcal{P}$ *is an expected value of a monomial over the random variables* x_i *of* $\mathcal{P}$.

Using Definition 5.3, in Algorithm 2 we compute $\mathbb{E}$-variables based on expected values $\mathbb{E}[x_i(n)]$ of loop variables x_i, as well as using higher-order and mixed moments of $\mathcal{P}$, such as $\mathbb{E}[x_i^k(n)]$

or $\mathbb{E}[x_i x_j(n)]$ (lines 3 and 9 of Algorithm 2). To this end, Algorithm 2 resembles the approach of [BKS19] and extends it to handle Prob-solvable loops with dependencies among random variables drawn from Gaussian/uniform distributions (line 9 of Algorithm 2). More specifically, Algorithm 2 uses *moment-based recurrences over* $\mathbb{E}$*-variables* from [BKS19], describing the expected values $\mathbb{E}[x_i(n)]$ of x_i as functions of other $\mathbb{E}$-variables (line 2 of Algorithm 2). To this end, note that Prob-solvable loop updates from (5.2)-(5.3)

over x_i yield linear recurrences with constant coefficients over $\mathbb{E}[x_i(n)]$, by using the following simplification rules over $\mathbb{E}$-variables:

$$\begin{array}{lcll} \mathbb{E}[expr_1 + expr_2] & \to & \mathbb{E}[expr_1] + \mathbb{E}[expr_2] & \\ \mathbb{E}[expr_1 \cdot expr_2] & \to & \mathbb{E}[expr_1] \cdot \mathbb{E}[expr_2], & \text{if } expr_1, expr_2 \text{ are independent} \\ \mathbb{E}[c \cdot expr_1] & \to & c \cdot \mathbb{E}[expr_1] & \\ \mathbb{E}[c] & \to & c & \\ \mathbb{E}[\mathcal{D} \cdot expr_1] & \to & \mathbb{E}[\mathcal{D}] \cdot \mathbb{E}[expr_1] & \end{array} \tag{5.4}$$

where $c \in \mathbb{R}$ is a constant, $\mathcal{D}$ is a known independent distribution, and $expr_1$, $expr_2$ are polynomial expressions over random variables.

Yet, to address our Prob-solvable loop extensions compared to [BKS19], in addition to (5.4) we need to ensure that dependencies among the random variables of $\mathcal{P}$ yield also moment-based recurrences. We achieve this by introducing the following two simplification rules over random variables with Gaussian/uniform distributions:

$$\begin{array}{lcl} \mathcal{G}(expr_\mu, \sigma^2) & \to & expr_\mu + \mathcal{G}(0, \sigma^2), \\ \mathcal{U}(expr_l, expr_u) & \to & expr_l + (expr_u - expr_l)\mathcal{U}(0, 1), \end{array} \tag{5.5}$$

for arbitrary polynomial expressions $expr_\mu, expr_l, expr_u$ over random variables. Using (5.5) in addition to (5.4), moment-based recurrences of Prob-solvable loops can always be computed as linear recurrences with constant coefficients over $\mathbb{E}$-variables (line 9 of Algorithm 2), implying the existence of closed form solutions of $\mathbb{E}$-variables and hence of MBIs of $\mathcal{P}$, as formalized below in Theorem 5.1.

Lastly, we introduce simplification rules that simplify the treatment of discrete random variables. These rules are based on the Theorem 5.3 and have the following form:

$$d^n \quad \to \quad \textstyle\sum_{i=0}^{m-1} a_i d^i, \tag{5.6}$$

where $m = |\Omega(d)|$ and a_is are constants as of Theorem 5.3. With these rules, the powers for the discrete part of $\mathbb{E}$-variable monomials never exceed a certain bound.

Theorem 5.1. *The simplification rules* (5.5) *are correct.*

Proof. Recall that there is a one-to-one correspondence between probability distributions and characteristic functions $E[e^{itX}]$ of a random variable X. In particular, the characteristic function of a Gaussian distribution with parameters μ and σ^2 is $\mathbb{E}[e^{it\mathcal{G}(\mu,\sigma^2)}]$ and thus the characteristic function of $\mathcal{G}(expr_\mu, \sigma^2)$ is $\mathbb{E}[e^{it\mathcal{G}(expr_\mu,\sigma^2)}]$. Then

$$
\begin{aligned}
\mathbb{E}\left[e^{it\mathcal{G}(expr_\mu,\sigma^2)}\right] &= \int \mathbb{E}\left[e^{it\mathcal{G}(y,\sigma^2)}f(y)\right]dy \\
&= \iint e^{itx}\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(x-y)^2}{2\sigma^2}}f(y)dxdy \\
&= \iint e^{it(x+y)}\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{((x+y)-y)^2}{2\sigma^2}}f(y)dxdy \\
&= \int e^{itx}\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(x)^2}{2\sigma^2}}dx\int e^{ity}f(y)dy \\
&= \mathbb{E}\left[e^{it\mathcal{G}(0,\sigma^2)}\right]\cdot\mathbb{E}\left[e^{it\cdot expr_\mu}\right] \\
&= \mathbb{E}\left[e^{it\left(\mathcal{G}(0,\sigma^2)+expr_\mu\right)}\right]
\end{aligned}
$$

by change of limits for $x \in \mathbb{R}$, where f is the probability density function of the random variable $expr_\mu$. Note that $\mathbb{E}\left[e^{it\left(\mathcal{N}(0,\sigma^2)+expr_\mu\right)}\right]$ corresponds to the characteristic function of $expr_1 + \mathcal{G}(0,\sigma^2)$, and hence the simplification rule $\mathcal{G}(expr_\mu,\sigma^2) \to expr_1 + \mathcal{G}(0,\sigma^2)$ of (5.5) is correct.

Similarly, for a uniform distributions with limits l, u we have the corresponding characteristic function $\mathbb{E}[e^{it\mathcal{U}(l,u)}]$, hence for $\mathcal{U}(expr_l, expr_u)$ we have

$$
\begin{aligned}
\mathbb{E}\left[e^{it\mathcal{U}(expr_l,expr_u)}\right] &= \int f(y)\mathbb{E}\left[e^{it\mathcal{U}(y_l,y_u)}\right]dy \\
&= \int f(y)\int_{y_l}^{y_u}\frac{1}{y_u-y_l}e^{itx}dxdy \\
&= \int f(y)\frac{1}{y_u-y_l}\int_0^1 (y_u-y_l)e^{it(y_l+(y_u-y_l)x)}dxdy \\
&= \int f(y)\int_0^1 1\cdot e^{it(y_l+(y_u-y_l)x)}dxdy \\
&= \int f(y)\mathbb{E}\left[e^{it(y_l+(y_u-y_l)\mathcal{U}(0,1))}\right]dy \\
&= \mathbb{E}\left[e^{it(expr_l+(expr_u-expr_l)\mathcal{U}(0,1))}\right]
\end{aligned}
$$

by the rules of expectations and the change of limits, with $y = (y_l, y_u)$ and f the probability density function of the random variable $(expr_l, expr_u)$. Expression $\mathbb{E}\left[e^{it(expr_l+(expr_u-expr_l)\mathcal{U}(0,1))}\right]$ corresponds to the characteristic function of $expr_l + (expr_u - expr_l)\mathcal{U}(0,1))$, so the simplification rule $\mathcal{U}(expr_1, expr_2) \to expr_1 + (expr_2 - expr_1)\mathcal{U}(0,1)$ of (5.5) is correct.

Further, observe that polynomial expressions remain polynomial after applications of (5.5) (line 9 of Algorithm 2). Once Gaussian and uniform distributions depending on loop variables are replaced using (5.5), we are left with independent known distributions and polynomial expressions over random variables for which (5.4) and (5.6) can further be used. □

Since the left- and right-hand sides of the rules (5.4), (5.5), and (5.6) are equal, in the mathematical sense, the MBIs computed correspond to the properties of the original loop $\mathcal{P}$.

We next prove, that arbitrary moments for variables of our new, extended version of Prob-solvable loops are computable.

Theorem 5.2. *Let $\mathcal{P}$ be an extended Prob-solvable loop (as of Definition 5.1) with discrete finite variables variables $\{d_1, \ldots, d_r\}$ and arbitrary variables $\{x_1, \ldots, x_m\}$ and consider $k \in \mathbb{N}$. Algorithm 2 is sound and terminating, yielding MBIs of degree k of $\mathcal{P}$.*

Proof. For the proof, we will use properties of C-finite recurrences, that is, linear recurrences with constant coefficients. Closed-form solutions always exist for homogeneous C-finite recurrences and are called C-finite expressions. Furthermore, any inhomogeneous C-finite recurrence can be translated into a homogeneous one if the inhomogeneous part is a C-finite expression.

We associate every monomial with an ordinal number as follows,

$$\prod_{i=1}^{r} d_i^{\delta_i} \prod_{i=1}^{m} x_i^{\lambda_i} \xrightarrow{\sigma} \sum_{i=1}^{m} \omega^m \cdot \lambda_i$$

and partially order monomials M, N such that $M > N$ iff $\sigma(M) > \sigma(N)$.

Algorithm 2 terminates if for every monomial M (from the set S, line 4 of Algorithm 2) the moment-based recurrence equation over the corresponding $\mathbb{E}$-variable $\mathbb{E}[M(n+1)]$ can be computed as a C-finite expression over $\mathbb{E}$-variables. We will show that this is indeed the case by transfinite induction over monomials.

Let $W = \prod_{i=1}^{K} x_i^{\lambda_i}$, $M = W \prod_{i=1}^{r} d_i^{\delta_i}$, $\mathcal{M} = \{W \cdot \prod_{i=1}^{r} d_i^{\gamma_i} | i \in \{1, \ldots, r\}, \forall i : \gamma_i < |\Omega(d_i)|\}$. Assume that every monomial smaller than M has a closed-form solution in the form of a C-finite expression.

Updates of $\mathcal{P}$ are of the form specified by equations (5.2)-(5.3) of Definition 5.1.

The algorithm rewrites $\mathbb{E}[M(n+1)]$ as a sum of $\mathbb{E}$-variables, which leads to

$$\mathbb{E}\Bigg[\prod_{i=1}^{r} \sum_{j} \sum_{\alpha}^{\alpha_{j,i}} p_{i,j,\alpha} \left(P_{i,j,\alpha}(d_1, \ldots, d_r)\right)^{\delta_i} \cdot \prod_{i=1}^{K} \sum_{j} \sum_{\beta}^{\beta_{j,k}} q_{k,j,\beta} \left(Q_{k,j,\beta}(d_1, \ldots, d_r) + R_{k,j,\beta(d_1,\ldots,d_r,x_1,\ldots,x_{k-1})}\right)^{\lambda_i} \Bigg]. \tag{5.7}$$

After applying the simplification rules, the expression simplifies to

$$\mathbb{E}[M(n+1)] = \sum_{M_i \in \mathcal{M}} c_{M_i} M_i(n) + \sum_{j=1}^{J} b_j \mathbb{E}[N_j(n)] \tag{5.8}$$

for some J, constants c_i, b_i, and monomials $N_1, \ldots, N_J$ all smaller than M. By assumption, there is a closed form expression for each N_j, hence for the sum $\sum_{j=1}^{J} b_j \mathbb{E}[N_j(n)]$. Since M

was arbitrary from $\mathcal{M}$, a similar recurrence can be derived for any monomial in $\mathcal{M}$. Since we further have $|\mathcal{M}| < \infty$, we have a system of $|\mathcal{M}|$ inhomogeneous linear C-finite recurrences of the form

$$\mathbb{E}[M^*(n+1)] = \sum_{M \in \mathcal{M}} c_{M^*,M} M(n) + f_{M^*}$$

for each $M^* \in \mathcal{M}$, and C-finite expressions f_{M^*}. Hence, the closed forms of $\mathbb{E}[M^*(n)]$ exist and are C-finite expressions. □

Example 5.1. *Consider the Prob-solvable loop in Figure 5.2(B). An example of* $\mathbb{E}$*-variable would be* $\mathbb{E}[W2^2]$*, for which an MBI* $\mathbb{E}[W2^2] = 4.01408a^2 + 53.83168a + 4.01408b + 250.3172$ *is computed using Algorithm 2.*

Remark. *While Prob-solvable loops are non-deterministic, with trivial loop guards of* $true$*, we note that probabilistic loops bounded by a number of iterations, such as*

$$n := 0; while(n < 1000)\{n := n + 1\}$$

can be encoded as Prob-solvable loops.

5.3 Encoding BNs as Prob-solvable Loops

In this section, we argue that Prob-solvable loops offer a natural way for encoding BNs, enabling further BN analysis via Prob-solvable loop reasoning in Section 5.4.

5.3.1 Finite Variables

For a finite discrete random variable X, let $[X = x]$ be the expression such that $[X = x] = 1$ if $X = x$ and 0 otherwise. Note that when X is binary-valued, we have $[X = 1] = X$ and $[X = 0] = 1 - X$. It follows that, in general, for a discrete variable X with possible values $x = 0, 1, \ldots, k-1$, we have $[X = x] = \prod_{\substack{0 \le i < k \\ i \neq d}} \frac{X-i}{x-i}$. Furthermore, let $[(X, Y) = (x, y)] = [X = x] \cdot [Y = y]$. Then, $[(X, Y) = (x, y)] = 1$ iff $X = x \wedge Y = y$, and 0 otherwise. Finally, we write $[X \neq x]$ to denote $1 - [X = x]$. Observe that $[X = x]$ and $[X \neq x]$ are polynomials in X.

Theorem 5.3. *Let* X *be a discrete random variable over* $A = \{a_1, \ldots, a_m\}$*. Then we can rewrite* X^n *as a linear combination of* $1, X, X^2, \cdots, X^m$*. Furthermore,*

$$X^n = \overline{a^n} M^{-1} \overline{X}, \tag{5.9}$$

where $\overline{a^n} = (a_1^n, \cdots, a_m^n)$*,* M *is an* $m \times m$ *matrix with* $M_{ij} = a_j^{i-1}$*, and* $\overline{X} = (X^0, \cdots, X^{m-1})^T$*.*

Proof. Since X has a value drawn from A, then $X - a = 0$ for some $a \in A$ thus $\prod_{a \in A}(X - a) = 0$. Hence also

$$X^k \prod_{a \in A} (X - a) = 0, \tag{5.10}$$

which gives

$$X^{k+m} = \sum_{0<i<m} d_i X^{k+i} \tag{5.11}$$

for some values d_i.

The sequence $1, X, X^2, \cdots$ then give rise to a recursively defined sequence $(x_k)_{k=0}^{\infty}$ such that $x_i = X^i$ defined by $x_k = X^K$ for $=\leq k < m$ and a recursive property as given by (5.10) and (5.11). We can now solve the recurrence, giving us another way of representing the sequence $1, X, X^2, \cdots$.

The characteristic polynomial of $(x_k)_{k=0}^{\infty}$ is $\prod_{a\in A}(X-a)$, thus we have

$$X^n = \sum_{0<i\leq m} c_i a_i^n,$$

with the coefficients c_i coming from the values of first m terms in the sequence, i.e. $x_0, x_1, \cdots, x_{m-1}$ or $1, X, \cdots, X^{m-1}$. This gives a system of linear equations

$$\begin{aligned} X^0 &= \sum_{0<i\leq m} c_i a_i^0 \\ &\vdots \\ X^{m-1} &= \sum_{0<i\leq m} c_i a_i^{m-1} \end{aligned}$$

or $M\overline{C} = \overline{X}$ where $\overline{X} = (X^0, \cdots, X^{m-1})^T$, $\overline{C} = (c_1, \cdots, c_m)^T$, and M with $M_{kl} = a_l^{k-1}$. Then we have $\overline{C} = M^{-1}\overline{X}$. Note that the values c_i are linear combinations of $X^0, \cdots, X^{m-1}$.

This gives

$$X^n = (a_1^n, a_2^n, \cdots, a_m^n) M^{-1} \overline{X}.$$

(Or $X^n = A^{\odot n} M^{-1}\overline{X}$ with Hadamard power notation.) □

This result allows us to rewrite any power of a finite random variable with m possible values in terms of its first $m-1$ moments.

Corollary 5.3.1. *For a binary random variable X we have $X^n = X$ for $n \geq 1$.*

The representation of arbitrary finite variables introduces polynomial dependencies, which were not supported previously. With the extension of Definition 5.1 and by Theorem 5.2, we can rewrite the $\mathbb{E}$-variables and the recurrences to guarantee the termination of Algorithm 2.

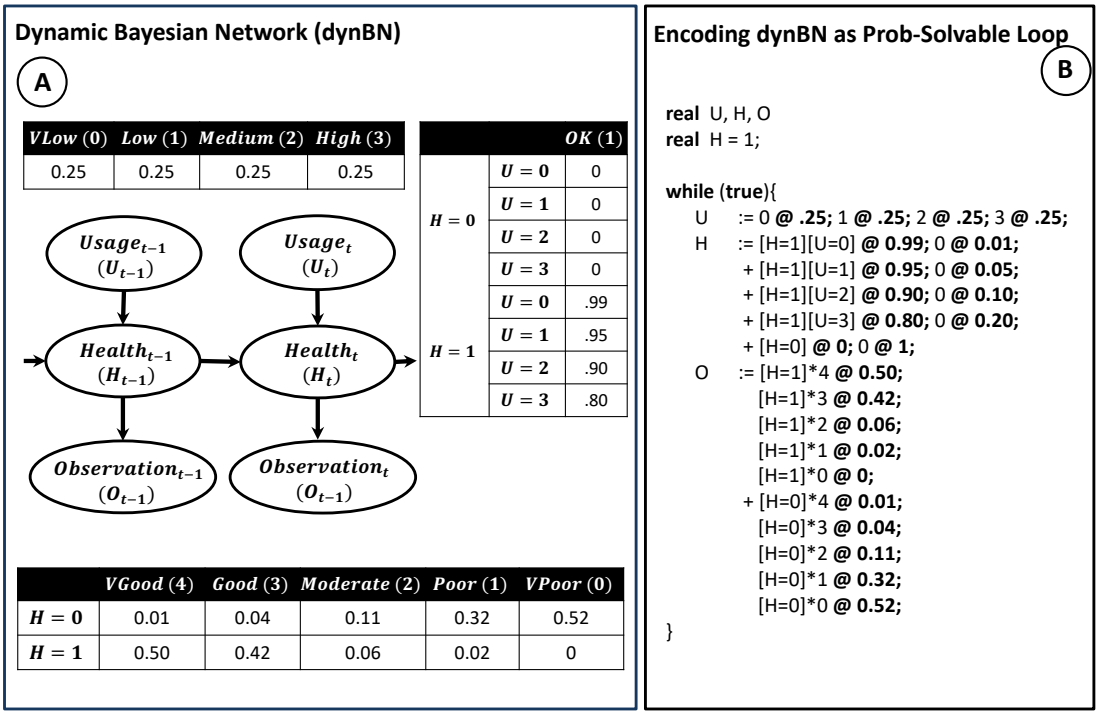

Figure 5.3: In Figure 5.3(B) we give the Prob-solvable loop encoding of the dynamic Bayesian Network (dynBN) from Figure 5.3(A).

Example 5.2. *With the Theorem 5.3, we are able to encode the BN of Figure 5.3. For the random variable usage over* $\{0,1,2,3\}$*, program variable* U*, we have*

$$\begin{aligned} X^n &= \begin{pmatrix} 0^n & 1^n & 2^n & 3^n \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \\ 0 & 1 & 4 & 9 \\ 0 & 1 & 8 & 27 \end{pmatrix}^{-1} \begin{pmatrix} X^0 \\ X^1 \\ X^2 \\ X^3 \end{pmatrix} \\ &= \begin{pmatrix} 0^n & 1^n & 2^n & 3^n \end{pmatrix} \begin{pmatrix} 1 & -11/6 & 1 & -1/6 \\ 0 & 3 & -5/2 & 1/2 \\ 0 & -3/2 & 2 & -1/2 \\ 0 & 1/3 & -1/2 & 1/6 \end{pmatrix} \begin{pmatrix} X^0 \\ X^1 \\ X^2 \\ X^3 \end{pmatrix}. \end{aligned} \tag{5.12}$$

With (5.12)*, we can reduce the power of* U *above three whenever it appears in the recurrence computation. Such higher power may appear when computing higher moments for random variables Health or Observation.*

5.3.2 Modeling Local Probabilistic Models of BNs as Prob-solvable Loop Updates

A BN is fully specified by its local dependencies. We consider common local probabilistic models and encode these models as Prob-solvable loop instances, as follows.

Deterministic Dependency

We first explore local probabilistic models specifying deterministic dependency, that is, when the values of BN nodes X are determined by the values of the parent variables from $Par(X)$. For example, when X is binary-valued, such a deterministic dependency can be a Boolean expression. On the other hand, when X is continuous, deterministic dependency can be a function over $Par(X)$.

For a continuous variable X whose value is given by a polynomial $Q(Par(X))$, encoding deterministic dependencies as a Prob-solvable loop update is straightforward: we simply set$X = Q(Par(X))$.

For a finite discrete random variable X, the representation as introduced in 5.3.1 provides a natural way to specify deterministic dependencies as updates of Prob-solvable loops (see Algorithm 3).

Conditional Probability Tables – CPTs

As shown in Figure 5.1(A), a common way to specify BN dependencies among discrete variables is with CPTs, with each CPT line representing a possible assignment of values of a BN node X to $Par(X)$. A CPT for X can be turned into Prob-solvable loop updates as follows.

We represent the values of X with integers $\{0, \cdots, m-1\}$ for an m-ary variable X. Let $Par(X) = \{Y_1, \cdots, Y_k\}$, denoting the parents of X. We represent each line L in the CPT for X by an update U_L. Each line L specifies the values for $Par(X)$. If L specifies values $Y_1 = y_1, \cdots, Y_k = y_k$ we let $D_L \equiv \bigwedge_i Y_i = y_i$ Let $P(X = i|L) = p_{L,i}$ and define U_L to be

$$0 \cdot [D_L] \mathbin{@} p_{L,0}; \cdots; (m-1) \cdot [D_L] \mathbin{@} p_{L,m-1}; \tag{5.13}$$

encoding that the value of U_L is 0 if D_L not as specified in the respective CPT line L; otherwise the value of X_L is i with probability $p_{L,i}$. We then set

$$X = \sum_{L \in CPT} X_L. \tag{5.14}$$

Example 5.3. *Using* (5.13)-(5.14)*, the disBN of Figure 5.1(A) is encoded as a Prob-solvable loop in Figure 5.1(B). While the parameters of S and G are not directly visible from the disBN, these parameters are given by the expected values of S and G in the Prob-solvable loop of Figure 5.1(B). Note that Figure 5.1(B) also features a GR variable corresponding to a Bernoulli random variable depending on G and R, such that GR is 1 iff both G and R are 1. The program variable $continue$ samples a sequence of Bernoulli random variables (one for each iteration n), while the random variable $count$ represents a geometric distribution encoding the sum of $continue$ values.*

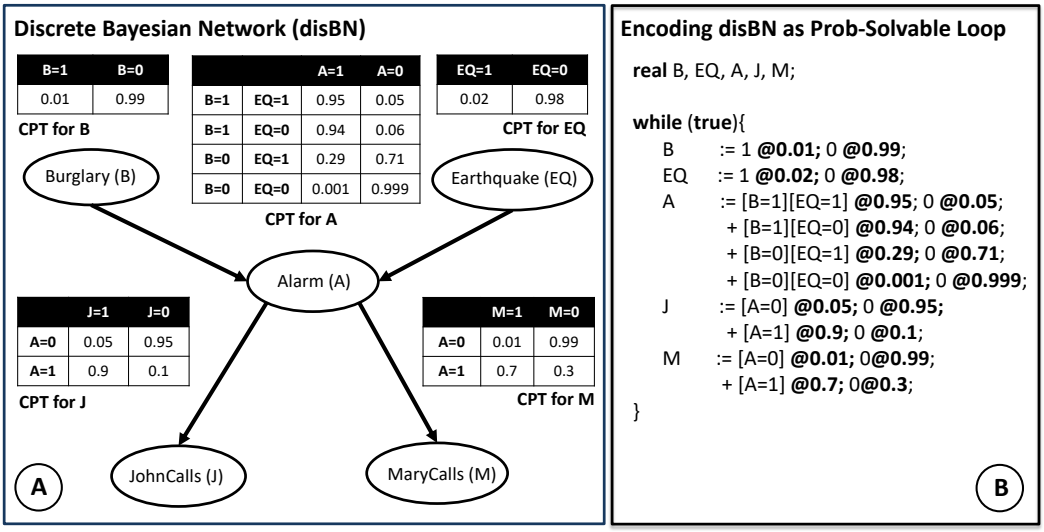

Figure 5.4: The discrete Bayesian Network (disBN) of Figure 5.4(A) shows a burglar alarm example. A burglar (B) and earthquake (EQ) directly affect the probability of the Alarm (A) going off, but whether or not John calls (J) or Mary calls (M) depends only on the alarm. A Prob-solvable loop encoding for this disBN is given in Figure 5.4(B).

Linear Dependency for Gaussian Variables

A local probabilistic model for a Gaussian random variable with continuous parents (as introduced in Definition 2.11) can be encoded as a Prob-solvable loops update, as follows:

$$X = RV\Big(gauss,\ \alpha_X + \sum_{Y \in Par(X)} \beta_{X,Y} \cdot Y,\ \sigma_X^2\Big), \tag{5.15}$$

where $\alpha_X, \beta_{X,Y}$ are constants, σ_X^2 is fixed and $RV(gauss, \mu, \sigma^2)$ denotes a Gaussian random variable drawn from a Gaussian distribution $\mathcal{G}(\mu, \sigma^2)$.

Conditional Linear Gaussian Dependency

By combining BN dependencies on discrete and continuous variables for a Gaussian random variable X, we can model conditional linear Gaussian dependencies for X. Let D be the joint distribution of the discrete parents of X and for each $d \in D$ let $\mathcal{G}_d$ be the Gaussian distribution associated with condition d (here $\mathcal{G}_d$ may depend on the values of continuous parents $Par(X)$ of X, as discussed in Section 5.2). The conditional linear Gaussian dependency for X can be modeled as the following Prob-solvable loop update:

$$\sum_{d \in \Omega(D)} [D = d] \cdot \mathcal{G}_d. \tag{5.16}$$

Example 5.4. *Figure 5.2(B) shows the Prob-solvable loop encoding of the clgBN of Figure 5.2(A). The random variables, $W1$ and $W2$ are given by conditional linear Gaussian dependency and*

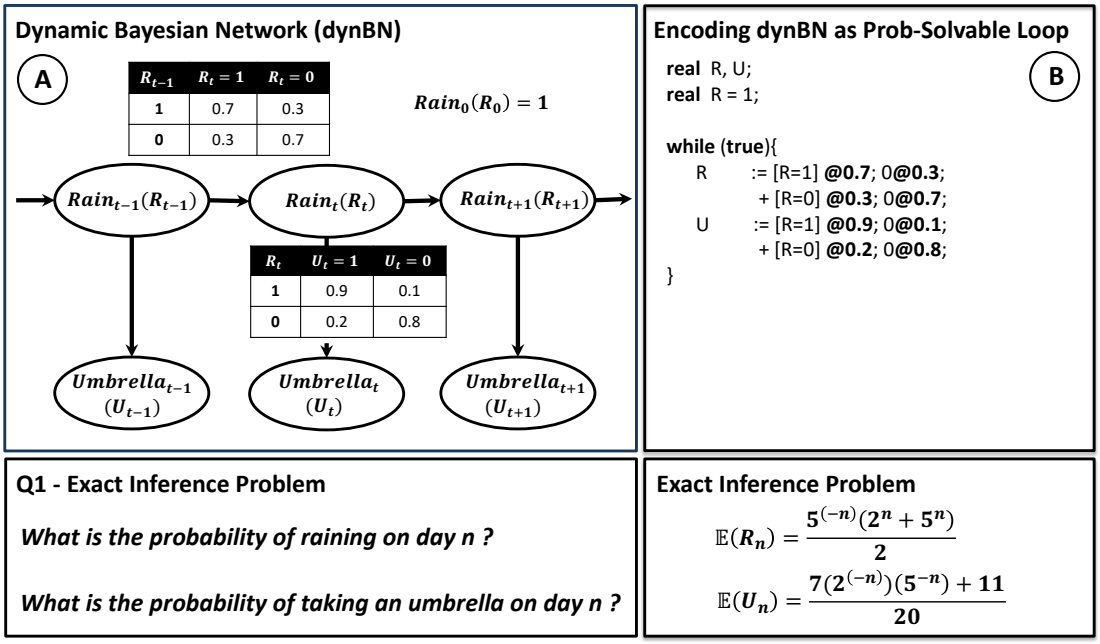

Figure 5.5: In Figure 5.5(B) we give the Prob-solvable loop encoding of the dynamic Bayesian Network (dynBN) from Figure 5.5(A). Solutions of probabilistic inferences in this dynBNs are also given, by computing MBIs of Figure 5.5(B).

encoded using (5.16). *For simplicity,* $W1$ *and* $W2$ *are further split into variables* $W1\_1$ *and* $W1\_2$*, and* $W2\_1$ *and* $W2\_2$*, respectively, representing different values of* $W1$ *and* $W2$ *based on the value of* D*. Further,* $D1\_S$ *is a binary variable which is* 1 *iff* D *is* 1 *and* S *is* 0*, and* $W2D1\_S$ *represents the expected value of* $W2 \cdot D1\_S$*.*

Temporal Dependencies in DynBNs

Dependencies in dynBNs are given by intra- and inter-time-slice edges. While the encoding of these dependencies is similar to the afore discussed BN dependencies, there are two restrictions on the structure of the dynBNs ensuring that dynBNs can be encoded as Prob-solvable loops. First, if X is not a finite discrete variable, its dependence on itself must be represented by a linear function. Second, a variable X can only depend on itself in the previous time-slice and current time-slice variables.

Example 5.5. *Figure 5.5(B) lists the Prob-solvable loop corresponding to Figure 5.5(A). The Bernoulli random variables* R *and* U *are encoded using* (5.13)-(5.14)*. The parameters of* R *and* U *change across iterations, corresponding to parameters in different time-slices of the dynBN; their concrete values are given by the expected values of* R *and* U*.*

Algorithm 3 Encoding BN variants as Prob-solvable loops

Input: BN
Output: Prob-solvable program
Notation: LPM denoting a local probabilistic model

$Nodes :=$ topologically ordered set of BN nodes
for X in $Nodes$ **do**
  if LPM of X is CPT **then**
    for each line L in the CPT **do** Set X_L as in (5.13)
    end for
    Set X as in (5.14)
  end if
  if LPM of X is a linear dependency for Gaussian variables **then** Set X as in (5.15)
  end if
  if LPM of X is a conditional linear Gaussian dependency **then** Set X as in (5.16)
  end if
end for

5.3.3 Encoding BNs as Prob-solvable Loops

Section 5.3.2 encoded common local probabilistic models of BN dependencies as Prob-solvable loop updates. Since BNs are DAGs, BN nodes can be ordered in such a way that each BN node X depends only on previous BN variables— its parents $Par(X)$. Hence, BNs can be encoded as Prob-solvable loops, as shown in Algorithm 3 and stated below.

Theorem 5.4. *Every BN and dynBN*[2] *with local probabilistic models given by CPT or (conditional linear) Gaussian dependencies can be encoded as a Prob-solvable loop. In particular, disBNs, gBNs, and clgBNs can be encoded as Prob-solvable loops.*

Based on Algorithm 3 and Theorem 5.4, we complete this section by defining the following class of BNs, in relation to Prob-solvable loops.

Definition 5.4 (Prob-solvable Bayesian Networks)**.** *A* Prob-solvable Bayesian Network (PSBN) *is a BN that can be encoded as a Prob-solvable loop.*

The relation and expressivity of PSBNs, and hence Prob-solvable loops, compared to BN variants is visualized in Figure 5.6.

5.4 Automatic BN Analysis via Prob-solvable Loop Reasoning

We now show that several BN challenges can automatically be solved by generating moment-based invariants of Prob-solvable loops encoding the respective BNs. To this end, (i) we consider exact inference, sensitivity analysis, filtering, and computing the expected number of rejecting

[2] subject to the restriction on structure of dynBN as discussed in Section 5.3.2

samples in sampling-based BN procedures and (ii) formalize these BN problems as reasoning tasks within Prob-solvable loop analysis. We then (iii) encode BNs as Prob-solvable loop $\mathcal{P}$ using Algorithm 3 and (iv) generate moment-based invariants of $\mathcal{P}$ using Algorithm 2. We address steps (i)-(ii) in Sections 5.4.1-5.4.3, and report on the automation of our work in Section 5.5.

5.4.1 Exact Inference in BNs

Common queries on BN properties address (i) the probability distributions of BN nodes X, for example, by answering whether $P(X = x)$ or $P(X < c)$; (ii) the conditional probabilities of BN nodes X, Y, such as $P(X = x|Y = y)$; or (iii) the expected values and higher-order moments of BN nodes X, Y, for instance, $\mathbb{E}[X], \mathbb{E}[X^2], \mathbb{E}[X|Y = y]$ and $\mathbb{E}[X^2|Y = y]$. Here we focus on (iii) but show that, in some BN variants, queries related to (ii) can also be solved by our work.

Exact Inference in disBNs

In the case when a BN node X is binary-valued, we have $\mathbb{E}[X] = P(X = true)$. Furthermore, for any higher-order moment of X we also have $Mom_k[X] = P(X = true)$. For non-binary-valued but discrete BN node X, with values from $\{0, \ldots, m-1\}$, the higher-order moments of X are also computable. Moreover, the first $m-1$ moments are sufficient to fully specify probabilities $P(X = i)$, for $i \in \{0, \ldots, m-1\}$, as shown in Theorem 5.3.

Lemma 5.5. *The higher-order moments of a discrete random variable X over $\{0, \ldots, m-1\}$ are specified by the first $m-1$ higher-order moments of X.*

Proof. Let $P(X = i) = p_i$, for $i \in \{0, \ldots, m\}$. Then $\sum_{0 \leq i < m} i^k p_k = Mom_k(X)$, yielding m linear equations over $p_0, \cdots p_{m-1}$, with $k \in \{1, \cdots m-1\}$. As $\sum_{0 \leq i < m} p_i = 1$, we have a linear system of m linearly independent equations, implying the existence of a unique solution that specifies the distribution of X. □

For computing conditional expected values and higher-order moments, we show next that deriving $\mathbb{E}[X^k|D = i]$ is reduced to the problem of computing $\frac{\mathbb{E}[X^k \cdot [D=i]]}{\mathbb{E}[[D=i]]}$.

Lemma 5.6. *If $D = i$ with non-zero probability, we have*

$$\mathbb{E}[X^k|D = i] = \frac{\mathbb{E}[X^k \cdot [D = i]]}{\mathbb{E}[[D = i]]}.$$

Proof. By partition properties for expected values, we have

$$\mathbb{E}[X^k[D = i]] = \mathbb{E}[X^k[D = i]|D = i]P(D = i) + \mathbb{E}[X^k[D \neq i]|D = i]P(D \neq i).$$

As $[D = i] = 1$ if and only if $D = i$, we derive $\mathbb{E}[X^k[D = i]|D = i] = \mathbb{E}[X^k|D = i]$ and $\mathbb{E}[X^k[D \neq i]|D = i] = 0$. Therefore, $\mathbb{E}[X^k|D = i] = \mathbb{E}[X^k|D = i]P(D = i)$. Since $P(D = i) \neq 0$, we conclude $\mathbb{E}[X^k|D = i] = \frac{\mathbb{E}[X^k \cdot [D=i]]}{\mathbb{E}[[D=i]]}$. □

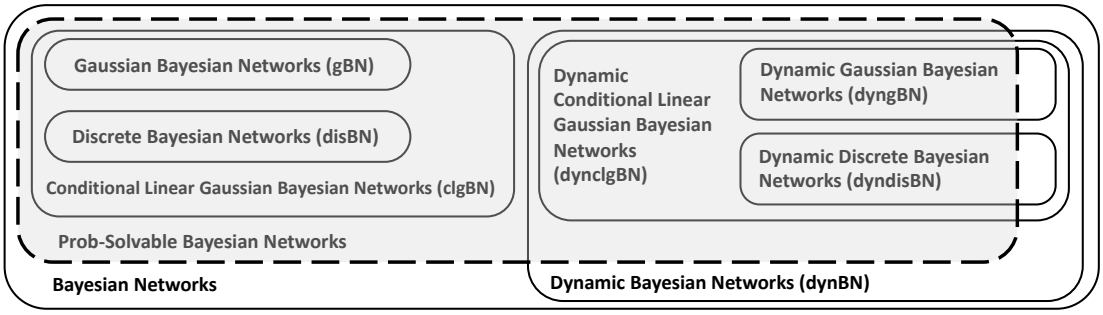

Figure 5.6: BN hierarchy.

Exact Inference in gBNs

Recall that a Gaussian distribution is specified by its first two moments that is, by its mean μ and variance σ^2. As all nodes in a gBN are Gaussian random variables, the first two moments of gBN nodes are sufficient to analyse gBN behavior. Further, $\mathbb{E}[X]$ and $\mathbb{E}[X^2]$ of a gBN node X are computable using Algorithm 2.

Exact Inference in clgBNs

As continuous variables X in clgBNs are Gaussian random variables, the means, and variance of X are also computable using Algorithm 2. However, clgBNs might also include discrete variables D, whose (conditional) higher-order moments can be computed as in Lemmas 5.5-5.6. Further, for a continuous variable X and a discrete variable D in a clgBN, we have

$$\mathbb{E}[X|D=i] = \frac{\mathbb{E}[X^k \cdot [D=i]]}{\mathbb{E}[[D=i]]},$$

allowing us, for example, to derive $\mathbb{E}[W2|D=1] = 7.25 + 0.89a$ in Figure 5.2.

Exact Inference in dynBNs

As dynBNs are infinite in nature, (infinite) Prob-solvable loops are suited to reason about dynBN inferences, such as (i) long-term behavior or prediction and (ii) filtering and smoothing. A related problem is characterizing the dynBN behavior after n iterations, in particular as $n \to \infty$.

(i) Prediction and long-term behavior in dynBNs By modeling dynBNs as Prob-solvable loops, we can compute/predict higher-order moments $\mathbb{E}[X_n^k]$ of dynBN nodes X using Algorithm 2, for an arbitrary n. Further, thanks to the existence of $\mathbb{E}[X_n^k]$ for Prob-solvable loops, we conclude that $\lim_{n\to\infty} \mathbb{E}[X_n^k]$ is also computable. Moreover, Algorithm 2 computes higher-order moments/MBIs in $O(1)$ time w.r.t. n, which is not the case of the $O(n)$ approach of the standard Forward algorithm.

(ii) Filtering and prediction in dynBNs Predicting the next dynBNs state X_{t+1} given the observations $e_1, \ldots, e_{t+1}$ until time $t+1$ can be expressed as $P(X_{t+1}|e_1, \ldots, e_{t+1})$, which in turn can be rewritten using Bayes' rule under the sensor Markov assumption (the evidence e_t depends only on program variables X_t from the same time-slice), as follows:

$$P(X_{t+1}|e_1, \ldots, e_{t+1}) = P(e_{t+1}|X_{t+1}) \cdot \sum_{x_t} P(X_{t+1}|x_t) \cdot P(x_t|e_1, \ldots, e_t),$$

where $P(e_{t+1}|X_{t+1})$ and $P(X_{t+1}|x_t)$ are specified by the BN, assuming discrete-valued observation variables. Filtering and prediction in dynBNs are thus computable using MBIs of Prob-solvable loops.

5.4.2 Number of BN Samples until Positive BN Instance

As pointed out in [BKKM18], an interesting question about BNs is "Given a Bayesian network with observed evidence, how long does it take in expectation to obtain a single sample that satisfies the observations?". A related though arguably simpler question would require giving the expected number of positive instances (samples satisfying the observation) in N samples of BNs. Both of these questions can be answered using standard results from probability theory.

Lemma 5.7. *Given the probability p of a BN observation, the expected number of positive BN instances in N samples is pN. Further, the expected number of BN samples until the first positive BN instance is $\frac{1}{p}$.*

Proof. Since every BN iteration (sample) is independent of previous ones, the occurrence of positive BN instances can be modeled as a Bernoulli random variable, given by the probability p of positive instances in any given iteration (or sample). Therefore, the number of positive instances in N samples is the sum of independent, identically distributed Bernoulli random variables, parametrized by p, following thus a Binomial distribution with parameters N and p. The number of positive BN samples is thus $\mathbb{E}[Binom(N,p)] = pN$. The expected number of BN samples until the first positive BN instance is thus given by the distribution of the number of Bernoulli trials needed for one success, which in turn is given by the geometric distribution $Geometric(p)$. The expected number of samples until the first positive BN instance is thus $\mathbb{E}[Geometric(p)] = \frac{1}{p}$. □

We note that Lemma 5.7 can be answered using Prob-solvable loop reasoning by relying on Algorithm 2, as illustrated next.

Example 5.6. *For inferring the expected number of positive instances in N samples in Figure 5.1, we first encode the observation in the BN as a new variable $GR = G \cdot R$, capturing the observation that the grass is wet and there was rain. We then transform the BN into a dynBN adding an inter-time-slice counter update $count = count + GR$. The expected number of positive instances is then the prediction $\mathbb{E}[count_n]$ for $n = N$.*

For answering the question of [BKKM18], we again encode the observation first as above, e.g., $GR = G \cdot R$. We use a boolean variable to indicate whether there has been a positive instance

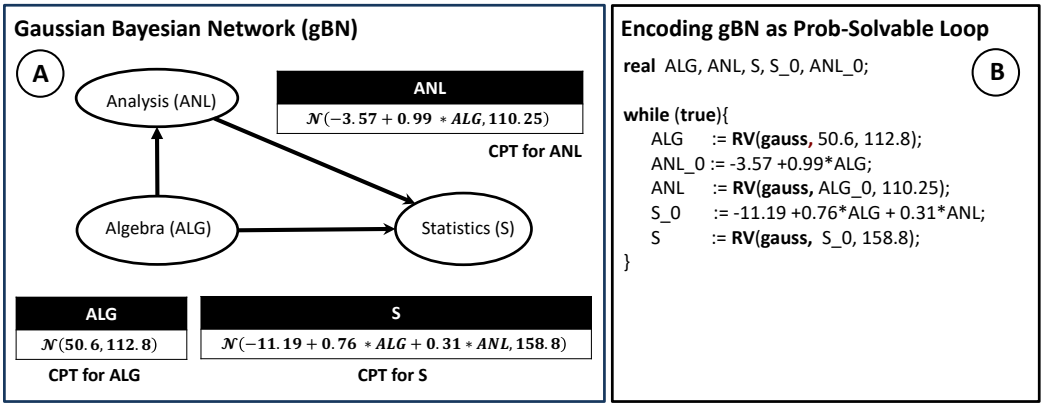

Figure 5.7: The Gaussian Bayesian Network (gBN) of Figure 5.7(A) describes the relationships between the marks on three math-related topics. Its respective Prob-solvable loop encoding is given in Figure 5.7(B).

$continue = continue \cdot [GR = 0]$, which is initiated as 1 (or true) and updated to 0 once $GR = 1$ and stays 0 thereafter. Finally, we update a loop counter as long as there is no positive instance observed with $count = count + continue$. The expected number of samples until the first positive instance is the long-term behavior of $count$, i.e., $\lim_{n\to\infty} \mathbb{E}[count_n]$.

5.4.3 Sensitivity Analysis in BNs

As BNs rely on network parameters, a challenging task is to understand to what extent can a small change in a network parameter influence the outcome of a particular BN query. This task is referred to as sensitivity analysis in BNs. More precisely, we would like to compute $P(X|e)$ and $\mathbb{E}[X|e]$ for a random variable X and evidence e as functions of a BN parameter(s) θ. For doing so, we note that Prob-solvable loops may use symbolic coefficients. Thus, replacing concrete BN probabilities with symbolic parameters and solving BN queries as discussed in Section 5.4.1, allow us to automate sensitivity analysis in BNs by computing MBIs of the respective Prob-solvable loops, using Algorithm 2.

Example 5.7. *A sensitivity analysis in Figure 5.2 could measure the effect of parameters of weight loss in week 1 on the conditional expectation $\mathbb{E}[W2|D = 1]$. That is, we compute $\mathbb{E}[W2|D = 1]$ as a function of parameters of $W1$. In this case, we introduce symbolic parameters a and b, adjusting the parameters of weight loss in week 1 ($W1\_1$) when the drug is administered. Using Algorithm 2, we compute the MBIs $\mathbb{E}[W2^k \cdot D], \mathbb{E}[D]$, from which we have, for $k = 1$, that $\mathbb{E}[W2|D = 1] = \frac{\mathbb{E}[W2 \cdot D]}{\mathbb{E}[D]} = 0.89a + 7.25$, answering the respective sensitivity analysis of Figure 5.2.*

5.5 Implementation and Experiments

We automated BN analysis via Prob-solvable loop reasoning by extending and using our tool MORA [BKS20b]. Although Algorithm 3 describes a way how to automatically translate BNs

Grass – Fig. 5.1 (disBN) #nodes: 3, #edges: 3, #parameters: 7		
Q1: $P(R\|G)$	$0.11s$	$P(R\|G) = \frac{1599}{1827} \approx 0.8752$
Q2: Number of samples till $G = 1$	$0.27s$	$\#samples = \frac{2500}{1827} \approx 1.37$
Q3: Sensitivity of $P(R\|G)$	$0.25s$	$P(R\|G) = \frac{0.04b+0.6396}{-0.178a+0.04b+0.7308}$
Q3: Number of samples sensitivity	$0.32s$	$\#samples = \frac{1}{-0.178a+0.04b+0.7308}$
Alarm [RN10] – Fig. 5.4 (disBN) #nodes: 5, #edges: 4, #parameters: 10		
Q1: $P(B\|A)$	$0.13s$	$P(B\|A) = \frac{470010}{1258221} \approx 0.3736$
Q1: $P(EQ\|M)$	$0.15s$	$P(EQ\|M) = \frac{1052777*20}{586817249} \approx 0.03588$
Q1: $P(\neg EQ \wedge \neg B\|A \wedge J)$	$0.65s$	$P(\neg EQ \wedge \neg B\|A \wedge J) = \frac{4486509}{11323989} \approx 0.3962$
Q1: $P(EQ \wedge \neg B\|M \wedge J)$	$0.46s$	$P(EQ \wedge \neg B\|M \wedge J) = \frac{365743899}{2084100239} \approx 0.1754$
Q2: Number of samples (for $M \wedge J$)	$0.37s$	$\#samples = \frac{10000000000000}{50054875461} \approx 19.98$
Q3: Sensitivity analysis (all of above)	$1.48s$	$P(B\|A) = \frac{10bq+940b}{-279bq+939b+289q+1}$ $P(EQ\|M) = \frac{45540bq+21010q}{-19251bq+64791b+19941q+1069}$ $P(\neg EQ \wedge \neg B\|A \wedge J) = \frac{bq-b-q+1}{-279bq+939b+289q+1}$ $P(EQ \wedge \neg B\|M \wedge J) = \frac{-366110bq+366110q}{-351261bq+1182201b+363851q+2259}$
Asia [LS88] (disBN) #nodes: 8, #edges: 8, #parameters: 18		
Q1: $P(Asia, Lung\|Dysp)$	0.46s	$P(A, L\|D) = \frac{439}{1547368} \approx 0.00028$
Q2: Samples till $Asia \wedge Lung$	1.23s	$\#samples = \frac{11}{20000} \approx 1818.19$
Q3: Sensitivity analysis	$1.18s$	$P(A, L\|D) = \frac{\frac{19a}{10000} + \frac{59b}{20000} + \frac{439}{2000000}}{-\frac{594a}{3125} - \frac{297b}{2500} + \frac{193421}{250000}}$
Marks [MKB79] – Fig. 5.7 (gBN) #nodes: 3, #edges: 3, #parameters: 6		
Q1: Marks: expected values	$0.02s$	$\mathbb{E}[ALG] = \frac{253}{5}, \mathbb{E}[ANL] = \frac{11631}{250}, \mathbb{E}[STAT] = \frac{1042211}{25000}$
Q3: Marks: sensitivity analysis EVs	$0.03s$	$\mathbb{E}[ALG] = \mu_{ALG}, \mathbb{E}[ANL] = \frac{99\mu_{ALG}}{100} - \frac{357}{100}, \mathbb{E}[STAT] = \frac{10669\mu_{ALG}}{10000} - \frac{122967}{10000}$
Q1: Marks: second moments	$0.05s$	$\mathbb{E}[ALG^2] = \frac{66829}{25}, \mathbb{E}[ANL^2] = \frac{149080491}{62500}, \mathbb{E}[STAT^2] = \frac{1272324089651}{625000000}$
Q3: Marks: sensitivity 2nd moments	$0.09s$	$\mathbb{E}[ALG^2] = \mu_{ALG}^2 + \frac{564}{5}$ $\mathbb{E}[ANL^2] = \frac{9801\mu_{ALG}^2}{10000} - \frac{35343\mu_{ALG}}{5000} + \sigma_{ANL} + \frac{6165009}{50000}$ $\mathbb{E}[STAT^2] = \frac{113827561\mu_{ALG}^2}{100000000} - \frac{1311934923\mu_{ALG}}{50000000} + \frac{961\sigma_{ANL}}{10000} + \frac{219203159849}{500000000}$
Q1: Average: expected values	$0.03s$	$\mathbb{E}[Avg] = \frac{4735311}{100000} \approx 47.35$
Q3: Average: sensitivity EV	$0.04s$	$\mathbb{E}[Avg] = \frac{40569\mu_{ALG}}{40000} - \frac{158667}{40000}$
Q1: Average: second moments	$0.08s$	$\mathbb{E}[Avg^2] = \frac{2380099013385 1}{10000000000} \approx 2380$
Q3: Average: sensitivity 2nd moment	$0.17s$	$\mathbb{E}[Avg^2] = \frac{1645843761\mu_{ALG}^2}{1600000000} - \frac{6436961523\mu_{ALG}}{800000000} + \frac{17161\sigma_{ANL}}{160000} + \frac{1133531965649}{8000000000}$
Rats [Edw12] – Fig. 5.2 (clgBN) #nodes: 4, #edges: 4, #parameters: 11		
Q1: $\mathbb{E}[W2\|D]$	$0.51s$	$\mathbb{E}[W2\|D] = 15.02$
Q3: $\mathbb{E}[W2\|D]$ sensitivity	$0.58s$	$\mathbb{E}[W2\|D] = 15.02 + 2.24a$
Q1: $\mathbb{E}[W2^2\|D]$	$0.99s$	$\mathbb{E}[W2^2\|D] = \frac{607089}{2500} = 242.8356$
Q3: $\mathbb{E}[W2^2\|D]$ sensitivity	$1.37s$	$\mathbb{E}[W2^2\|D] = \frac{3136}{625}a^2 + \frac{42057}{625}a + \frac{3136}{625}b + \frac{607089}{2500}$
win95pts (disBN) #nodes: 76, #edges: 112, #parameters: 574		
Q1: $\mathbb{E}[PrtPort\|DS\_LCLOK]$	3.08s	$= \frac{989900017424505}{999770762903873} \approx 0.99$
Q3: $\mathbb{E}[PrtPort\|DS\_LCLOK]$	4.38s	*(Full result omitted due to the size)*
Q1: $\mathbb{E}[PgOrnttnOK\|\neg Problem2]$	15.8s	$= \frac{19}{20}$
Q3: $\mathbb{E}[PgOrnttnOK\|\neg Problem2]$	17.7s	*(Full result omitted due to the size)*
Q1: $\mathbb{E}[GDIOUT \wedge \neg PrtFile\|PrtSel]$	160s	$= \frac{5000324762761511009932 01}{2000000000000000000000000} \approx 0.25$
Q3: $\mathbb{E}[GDIOUT \wedge \neg PrtFile\|PrtSel]$	182s	*(Full result omitted due to the size)*
Q1: $\mathbb{E}[PrtStatPaper \wedge LclOK\|Problem6 \wedge \neg PrtFile]$	339s	*(Full result omitted due to the size)*
Q3: $\mathbb{E}[PrtStatPaper \wedge LclOK\|Problem6 \wedge \neg PrtFile]$	445s	*(Full result omitted due to the size)*

Table 5.1: BN analysis via Prob-solvable loop reasoning within MORA.

to Prob-solvable loops, this part of the process was not implemented for the experiments below due to the lack of standard encoding for (dynamic) Bayesian networks. Bayesian networks were first manually encoded as Prob-solvable loops following the Algorithm 3. We then extended MORA to support our extended programming model of Prob-solvable loops and integrated

Algorithm 2 within MORA[3] to generate MBIs of Prob-solvable loops, thus solving the BN problems of Sections 5.4.1-5.4.3. As benchmarks, we used 41 BN-related problems for 8 BNs taken from [Edw12, MKB79, KN10, LS88, RN10, Mod19]. Tables 5.1 and 5.2 summarize our experiments. For each example in Tables 5.1-5.2, we list the BN queries we considered, that is, probabilistic inference (Q1), number of BN samples (Q2), and sensitivity analysis (Q3) as introduced in Section 4.1 and discussed in Sections 5.4.1-5.4.3. Column 2 shows the time needed by MORA to compute moment-based invariants (MBIs) solving the respective BN problems. The last column gives our derived solutions for the considered BN queries. Computation times are improved compared to [BKS20a]. This is partially thanks to code optimization in the tool implementation, unrelated to work presented in the paper. The most significant improvement, however, is seen in the discrete BNs. In this case, our improved timing results are mainly due to the fact that the number of auxiliary variables is reduced in the extended framework, reducing the amount of algebraic manipulation and recurrence solving. For example, the inference of $\mathbb{E}[PrtPort|DS\_LOCK]$ in the original form (with auxiliary variables) takes $67s$ in the approach of [BKS20a]. In contrast, our current work computes results within $3s$ after performing rewriting into a single, larger assignment. Our experiments were run on a MacBook Pro 2017 with 2.3 GHz Intel Core i5 and 8GB RAM.

Limitations Let us address some of the limitations of our approach and the implementation. As Figure 5.6 illustrates, our approach does not work for arbitrary (dynamic) Bayesian networks. The limitations arise from the restrictions we impose on the dependencies within the network and from the limits of the encodings specified in Section 5.3.

Some of the restrictions are *strong*, in the sense that they keep us within the algebraic frameworks introduced with our approach. Amongst strong restrictions are, for example, the restriction on polynomial dependence of (unbounded) variables on themselves or limiting discrete variables to only depend on discrete variables. The former guarantees the finiteness of the system of recurrences we need to solve, while the latter ensures we stay within the realm of polynomial arithmetic.

In contrast, we believe our *weak* restrictions do not violate any of the algebraic methods we use. An example would be the restriction on inter-time-slice dependencies of variables in DBNs. Many of the weak restrictions arise from the richness of the world of Bayesian networks. There are too many ways to specify conditional probability dependencies (CPDs) to handle them all, such as deterministic CPDs or tree-CPDs, and a variety of distributions can be used (other than the ones introduced in the paper).

Complexity Our approach relies on algebraic techniques (such as finding roots of polynomials and solving recurrences), for which external libraries are used (sympy, diofant), and hence giving precise overall complexity bounds is challenging. We shall further discuss some partial complexity results of our algorithms.

[3] https://github.com/probing-lab/mora

Encoding Bayesian networks: The encoding of a BN as a probabilistic loop is linear (in the size of BN) in both memory and time. A single probabilistic branch is used for each line of a table in discrete and CLG dependencies.

Encoding of the probabilistic queries: Each of the queries considered in our work can be encoded using only a constant number of new variables and corresponding polynomial probabilistic updates. The size of the polynomials is, in the worst case, exponential in the number of variables that appear in the query.

Number of recurrences: The number of recurrences that need to be constructed and solved depends on the query and is, in the worst case, exponential (in the number of variables). Compared to [BKS20a], only a single program variable is needed to encode any CPT, reducing the number of recurrences involving a given variable exponentially (in the number of parents of the given variable). In BNs with variables with many parents, this leads to significant improvement (e.g., from $67s$ to $3s$ for the inference query $\mathbb{E}[PrtPort|DS\_LOCK]$ in the *win95pts* network). The exponential time complexity corresponds to the NP-hardness of exact inference in BNs.

Comparison Our algebraic methods allow us to work with symbolic constants to analyze entire classes of BNs as a single problem and provide sensitivity analysis, encode various distributions and dependencies without the need to modify the entire approach, and address several different BN-related challenges within one framework. On the other hand, manipulating algebraic expressions and reasoning about algebraic objects is difficult and often slow. Single-purpose tools optimized for their task will likely perform better than the approach presented here.

Encoding a BN as a probabilistic program was also used in [BKKM18] to address computing the expected number of samples (Q2), where a syntactic fragment of pGCL called *Bayesian Network Langauge* (BNL) was used. Unlike Prob-solvable loops, BNL does not allow information flow across loop iterations and hence cannot model DBNs. Furthermore, only discrete variables are allowed in the BNL, further restricting the classes of BNs that can be modeled.

Approximate simulation-based methods, such as rejection sampling, are also often used to answer probabilistic queries on Bayesian networks. There are two main disadvantages of such approaches. First, events with very low probability require a great number of samples. We partially addressed this by computing the expected number of samples beforehand (Q2). Second, specifying local dependencies using symbolic parameters is not possible. Allowing symbolic parameters in Prob-solvable loops allows evaluating probabilistic queries on an entire class of BNs, as well as analyzing the sensitivity of BNs.

5.6 Chapter Conclusion

We show how to encode various kinds of Bayesian networks as an extended version of Prob-solvable loops [BKS19]. This new class of probabilistic programs supports polynomial arithmetic using finite-value variables as polynomial coefficients in polynomial updates. We provide a fully automatic approach that can compute, for this class of programs, closed-form expressions

Umbrella [RN10] – Fig. 5.5 (dynBN) #nodes: 2, #edges: 2, #parameters: 3		
Q1: Prediction	$0.44s$	$\mathbb{E}[R] = \frac{2^n+5^n}{2\cdot 5^n}$ $\mathbb{E}[U] = \frac{7\cdot 2^n}{20\cdot 5^n} + \frac{11}{20}$
Q1: Long-term behaviour	$0.44s$	$\mathbb{E}[R] \to \frac{1}{2}$ as $n \to \infty$ $\mathbb{E}[U] \to \frac{11}{20}$ as $n \to \infty$
Q3: Prediction - sensitivity	$0.47s$	$\mathbb{E}[R] = \frac{-3+10(-1+r)(-\frac{3}{10}+r)^n}{-13+10r}$ $\mathbb{E}[U] = \frac{(-47+70(-1+r)(-\frac{3}{10}+r)^n+20r)}{-130+100r}$
Q3: Long-term - sensitivity	$0.47s$	$\mathbb{E}[R] \to \frac{3}{13-10r}$ as $n \to \infty$ $\mathbb{E}[U] \to \frac{47-20r}{130-100r}$ as $n \to \infty$
Defect [Mod19] – Fig. 5.3 (dynBN) #nodes: 3, #edges: 3, #parameters: 17		
Q1: Prediction	$1.11s$	$\mathbb{E}[O=0] = \frac{13}{25} - \frac{13\cdot 91^n}{25\cdot 10^{2n}}$ $\mathbb{E}[O=1] = \frac{8}{25} - \frac{3\cdot 91^n}{10\cdot 10^{2n}}$ $\mathbb{E}[O=2] = \frac{11}{100} - \frac{91^n}{20\cdot 10^{2n}}$ $\mathbb{E}[O=3] = \frac{1}{25} + \frac{19\cdot 91^n}{50\cdot 10^{2n}}$ $\mathbb{E}[O=4] = \frac{1}{100} + \frac{49\cdot 91^n}{100\cdot 10^{2n}}$
Q1: Long-term behaviour	$1.11s$	$\mathbb{E}[O=0] \to \frac{13}{25}$ as $n \to \infty$ $\mathbb{E}[O=1] \to \frac{8}{25}$ as $n \to \infty$ $\mathbb{E}[O=2] \to \frac{11}{100}$ as $n \to \infty$ $\mathbb{E}[O=3] \to \frac{1}{25}$ as $n \to \infty$ $\mathbb{E}[O=4] \to \frac{1}{100}$ as $n \to \infty$
Q3: Prediction - sensitivity	$2.63s$	$\mathbb{E}[O=0] = \frac{(25ab-13a+91b)(25a+91)^n}{(25a+91)\cdot 10^{2n}} - b$ $-\frac{1183(25a+91)^n}{(625a+2275)\cdot 10^{2n}} + \frac{13}{25}$ $\mathbb{E}[O=3] = \frac{19a(25a+91)^n}{(50a+182)\cdot 10^{2n}} + \frac{1729(25a+91)^n}{(1250a+4550)\cdot 10^{2n}} + \frac{1}{25}$
Q3: Long-term - sensitivity	$2.63s$	$\mathbb{E}[O=0] \to \frac{13}{25}$ as $n \to \infty$ $\mathbb{E}[O=3] \to \frac{1}{25}$ as $n \to \infty$

Table 5.2: DBN analysis via Prob-solvable loop reasoning within MORA.

summarising the behavior of high-order moments of the program variables over infinite loops. This allows us to turn several BN analyses into the problem of computing moment-based invariants of Prob-solvable loops. In particular, we show how our approach can automate exact inference, sensitivity analysis, filtering, and computing the expected number of rejecting samples in sampling-based procedures via Prob-solvable loop reasoning.

CHAPTER 6

Further Developments

In this chapter, we give a brief overview of the latest research ([MSBK22, ABK+22, KMS+22b, KMS+22a]) building on the work of this thesis. The goal is to present the main results and the intuition or ideas behind them, so most of the technical details are omitted.

In Section 6.1, based on [MSBK22], we present the latest developments in the theoretical understanding of the class of programs that can be analyzed using methods of this thesis, the theory of *moment-computability*, and we give a precise characterization of such class of probabilistic loops, which we term *moment-computable*.

We then address the analysis of loops that are not moment-computable. In Section 6.2, based on [ABK+22], we study loops where a variable can admit a polynomial self-dependency, for which, in general, moment-based invariants cannot be computed automatically. We then consider loops with non-polynomial updates in Section 6.3, based on [KMS+22b].

Section 6.4, based on [KMS+22a], discusses the use of moment-based invariants to estimate distributions of program variables and the quality of such estimates.

[MSBK22] Marcel Moosbrugger, Miroslav Stankovic, Ezio Bartocci, and Laura Kovács. This is the moment for probabilistic loops. *Proc. ACM Program. Lang.*, 6(OOPSLA2):1497–1525, 2022.

[ABK+22] Daneshvar Amrollahi, Ezio Bartocci, George Kenison, Laura Kovács, Marcel Moosbrugger, and Miroslav Stankovic. Solving invariant generation for unsolvable loops. In *Static Analysis - 29th International Symposium, SAS 2022*, volume 13790 of *Lecture Notes in Computer Science*, pages 19–43. Springer, 2022. **Radhia Cousot Young Researcher Best Paper Award**.

[KMS+22b] Andrey Kofnov, Marcel Moosbrugger, Miroslav Stankovic, Ezio Bartocci, and Efstathia Bura. Moment-based invariants for probabilistic loops with non-polynomial

$lop \in \{and, or\}$, $cop \in \{=, \neq, <, >, \geq, \leq\}$, $Dist \in \{Bernoulli, Normal, Uniform, \dots\}$

⟨*sym*⟩ ::= a | b | ... ⟨*var*⟩ ::= x | y | ...

⟨*const*⟩ ::= $r \in \mathbb{R}$ | ⟨*sym*⟩ | ⟨*const*⟩ (+ | * | /) ⟨*const*⟩

⟨*poly*⟩ ::= ⟨*const*⟩ | ⟨*var*⟩ | ⟨*poly*⟩ (+ | − | *) ⟨*poly*⟩ | ⟨*poly*⟩**n

⟨*assign*⟩ ::= ⟨*var*⟩ = ⟨*assign_right*⟩ | ⟨*var*⟩ , ⟨*assign*⟩ , ⟨*assign_right*⟩

⟨*categorical*⟩ ::= ⟨*poly*⟩ ({ ⟨*const*⟩ } ⟨*poly*⟩)* [{ ⟨*const*⟩ }]

⟨*assign_right*⟩ ::= ⟨*categorical*⟩ | Dist (⟨*poly*⟩*) | Exponential (⟨*const*⟩ / ⟨*poly*⟩)

⟨*bexpr*⟩ ::= true (*) | false | ⟨*poly*⟩ ⟨*cop*⟩ ⟨*poly*⟩ | not ⟨*bexpr*⟩ | ⟨*bexpr*⟩ ⟨*lop*⟩ ⟨*bexpr*⟩

⟨*ifstmt*⟩ ::= if ⟨*bexpr*⟩ : ⟨*statems*⟩ (else if ⟨*bexpr*⟩ : ⟨*statems*⟩)* [else: ⟨*statems*⟩] end

⟨*statem*⟩ ::= ⟨*assign*⟩ | ⟨*ifstmt*⟩

⟨*statems*⟩ ::= ⟨*statem*⟩+

⟨*loop*⟩ ::= ⟨*statem*⟩* while ⟨*bexpr*⟩ : ⟨*statems*⟩ end

Figure 6.1: Grammar describing the syntax of probabilistic loops ⟨*loop*⟩.

assignments. In *Proc. of QEST 2022: Quantitative Evaluation of Systems - 19th International Conference*, volume 13479 of *Lecture Notes in Computer Science*, pages 3–25. Springer, 2022. **Best Paper Award**.

[KMS+22a] Ahmad Karimi, Marcel Moosbrugger, Miroslav Stankovic, Laura Kovács, Ezio Bartocci, and Efstathia Bura. Distribution estimation for probabilistic loops. In *Proc. of QEST 2022: Quantitative Evaluation of Systems - 19th International Conference*, volume 13479 of *Lecture Notes in Computer Science*, pages 26–42. Springer, 2022.

6.1 Moment-Computability

The methods developed in [BKS19] (Chapter 3) were further refined in [MSBK22]. The main results, which we cover in this section, are an extension to and a precise characterization of the class probabilistic loops for which moments of program variables can be computed.

Programming model The extended programming model is defined by the syntax as given in Figure 6.1. As programs considered in this thesis, the model allows loop-free initialization followed by a single loop. Unlike the model from Chapter 3, the loop body allows nested if-statements, arbitrary polynomial arithmetic, simultaneous assignments, and multiple assignments to the same variable. The model still supports drawing from common distributions, even continuous ones with unbounded support and symbolic constants.

Moment-computability The syntax is quite generous, and not for all programs allowed by this programming model can MBIs be computed. Whether we can compute MBIs for a given

program largely depends on the (polynomial) dependencies amongst variables present in the loop. We formalize what we mean by variable dependence as follows:

Definition 6.1 (Variable Dependency [MSBK22, Def. 15])**.** *Let $\mathcal{P}$ be a probabilistic loop and x, y be variables in $\mathcal{P}$. We define:*

- y depends conditionally on x*, if there is an assignment of y within an if-else-statement and x appears in the if-condition.*
- y depends finitely on x*, if x is finite and appears in an assignment of y.*
- y depends linearly on x*, if x appears only linearly in every assignment of y.*
- y depends polynomially on x*, if there is an assignment of y in which x appears non-linearly and x is not finite*
- y depends on x *if it depends on x conditionally, finitely, linearly, or polynomially.*

Furthermore, we consider the transitive closure for variable dependency as follows: If z depends on y and y depends on x, then z depends on x. If one of the two dependencies is polynomial, then z depends polynomially on x.

With this, we can formulate the main result of moment-computability: precise characterization of moment-computable loops.

Theorem 6.1 (Moment-Computability [MSBK22, Thm. 6])**.** *A probabilistic loop $\mathcal{P}$ is moment-computable if* (1) none of its non-finite variables depends on itself polynomially*, and* (2) if the variables in all if-conditions are finite.

Let us now briefly discuss the necessity of the two conditions. Even without considering probabilistic behaviour, removing any of the two restrictions leads to problems that are beyond possible.

(1) If the restriction on polynomial self-dependence is loosened, one can easily encode the *logistic map*, a first-order quadratic recurrence defined by $x_{n+1} = r \cdot x_n(1 - x_n)$. The logistic map is well-known for its chaotic behaviour, and, for most values r, has no analytical solution.

(2) Even without polynomial assignments, we can model a Turning machine once we are allowed if conditions over non-finite variables. One way to do that would be to model the tape using two variables l and r, whose binary representation represents the tape left and right from the tape. Writing and shifting the head can be done by addition/subtraction and multiplying by 2 or $\frac{1}{2}$, with a loop taking care of the remainder after the multiplication by $\frac{1}{2}$ (loosening the condition (2) is necessary for this). With a binary variable $term$ storing whether the Turing machine has terminated, computing the moment-based invariants for $term$ would solve the Halting problem. As the Halting problem is undecidable, condition (2) is necessary to guarantee the moment-computability.

6.2 Polynomial Self-dependencies

In [ABK+22], we consider programs with polynomial self-dependencies, i.e., programs violating condition (1) of Theorem 6.1. As argued in the previous section, (higher) moments of program variables cannot be, in general, computed. Instead, in this work, we investigate combinations of program variables for which solutions can be reached. A novel technique is presented that automatically synthesizes polynomials over program variables that admit closed-form solutions and thus leads to polynomial loop invariants.

This work, although arising from the analysis of probabilistic programs, is more general and advances the state-of-the-art in loop invariant generation for non-probabilistic programs as well.

Example 6.1. *Consider the following program.*

$x, y = 1, 1$
while $\star$ ***do***
  $w = x + y$
  $x = w^2$
  $y = w^3$
end while

Due to the polynomial dependencies, closed form solutions to the recurrences for variables x, y, and z cannot be easily computed. However, there is a polynomial invariant $y^2(n) - x^3(n) = 0$.

The technique for deriving loop invariants can be summarized as follows:

Step 1: Classify program variables as *effective* and *defective*. Intuitively, effective variables are those for which closed forms can be computed. Defective variables, on the other hand, are the sources of unsolvability and do not, generally, admit close-form solutions. More precisely, a variable is defective if it depends on itself polynomially or on a defective variable. Otherwise, it is effective. The notions of effectiveness/defectiveness are also extended to monomials. A monomial is defective if it contains a defective variable; otherwise, it is effective.

Step 2: For a fixed degree bound d we construct a *candidate polynomial* $S(n) = \sum c_W W$, where W are defective monomials in program variables of degree at most d and c_W are corresponding symbolic constants. Recurrence for $S(n)$ is then computed symbolically, in a similar fashion as the recurrences for E-variables in Chapter 3.

Step 3: Although variables in $S(n)$ are defective, $S(n)$ may admit a closed form. This happens if the recurrence is well-behaved, e.g., when it has the form

$$S(n+1) = \kappa S(n) + \sum c_M M, \tag{6.1}$$

where M are effective monomials and c_M unknown constats.

Step 4: The question then is: are there constants c_W, c_M, and κ that satisfy (6.1)? This leads to a system of (quadratic) equations. If such constants exist, this leads to a first-order linear recurrence for $S(n)$ for which a closed form can be computed. The closed form then leads to a polynomial invariant for the program loop.

Steps 2-4 can be repeated, increasing the degree bound d.

6.3 Non-polynomial Updates

Probabilistic programs with non-polynomial updates are addressed in [KMS+22b]. Updates with non-polynomial functions often arise from models with complex dynamics, such as the use of trigonometric functions to describe rotational movements. How can we leverage the advances in the analysis of PPS with polynomial arithmetic to compute MBIs of PPs with non-polynomial updates?

A method based on polynomial chaos expansion (PCE) is presented in [KMS+22b]. PCE approximates a random variable in terms of a polynomial in known random variables and can be applied when the second moment of the variable is finite. For any L^2 function $g : \mathbb{R}^k \to \mathbb{R}$, independent continuous random variables $z_1, \ldots, z_k$, and degree bound d we can approximate $g(z_1, \ldots, z_k)$ as a polynomial $\hat{g}$ in $z_1 \ldots, z_k$ of degree bounded by d. The coefficients of $\hat{g}$ depend on g itself and the orthonormal polynomial basis determined by $z_1, \ldots, z_k$.

We also address the quality of approximation in terms of the convergence rate with respect to the degree bound d. Under certain conditions, we achieve optimal, i.e., exponential, convergence rate. This is the case in *iteration stable* loops, i.e., loops the random variables $z_1, \ldots, z_k$ are continuous, independent, and identically distributed *across iterations* for every non-polynomial L^2-type update $g(z_1, \ldots, z_k)$. Optimal convergence can also be achieved when in *iteration non-stable* loops if we restrict our interest to finitely many loop iterations. For general *iteration non-stable* loops, we derive an upper bound on the approximation error.

Once the non-polynomial functions are replaced by their polynomial approximations, the program can be analysed using the methods from quantitative loop analysis, for example, the ones presented in this thesis.

6.4 Quantitative Evaluation

In [KMS+22a], we study techniques for distribution estimation from moments. We automate the process of estimation for methods based on Gram-Charlier expansion and Maximal Entropy. The 2 main questions addressed in the paper are:

1. *How to automatically estimate distributions of program variables of a PP from its moments?*
2. *Are the distribution estimations based on moments accurate?*

For the automated estimation, we assume that a probabilistic loop, as well as a set of moments, are given as input. We present two ways of estimating distributions. The first method is based on Gram-Charlier series expansion and can be used with parametric moments, in particular parametric with respect to loop iteration, i.e., in the form of MBIs. The second method is based on Maximal Entropy and tries to minimize information contained in the estimation while conforming to the moments given. In both cases, the estimation is given in the form of a probability density function (pdf).

For the purpose of evaluating the two methods, we also compute an estimation based on sampling, which we use as a proxy for the true distribution represented by the probabilistic program.

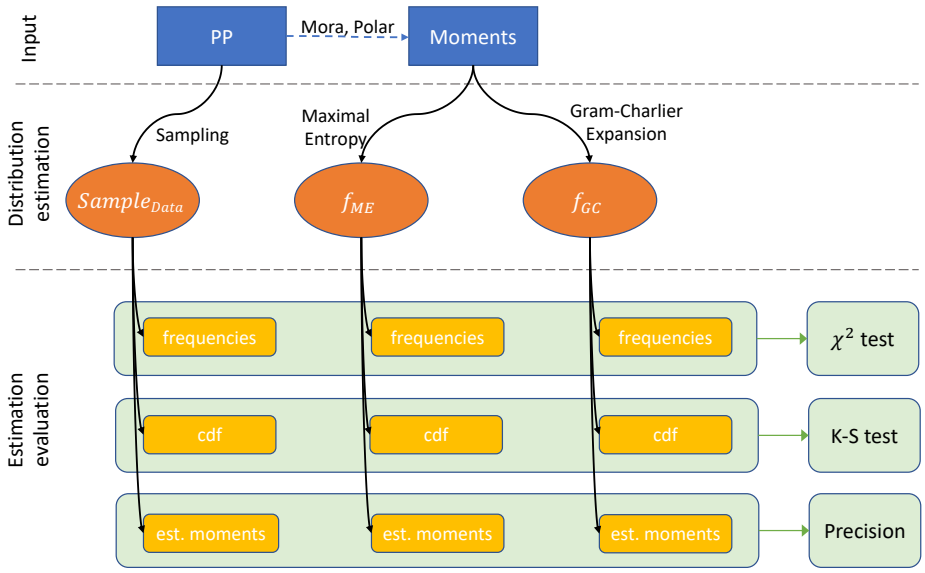

Figure 6.2: Overview of the process of distribution estimation and estimation evaluation.

The accuracy of the estimations is then evaluated in 3 ways, each requiring a slightly different representation of the distribution. The overview of the entire process is represented in Figure 6.2.

Chi-squared (χ^2) goodness-of-Fit test detects statistically significant differences between observed and expected frequencies. Expected frequencies are taken from the sample data, partitioned into intervals.Expected frequencies can be computed from estimated pdfs f_{GC} and f_{ME} from GC and ME estimation methods. For an interval I, the expected frequencies are $\int_I f_{GC}(x)dx$ (and similarly for ME). Test statistic measuring the deviation between expected and observed frequencies is computed and compared to the critical value determined by the number of samples and the number of intervals of partition. Based on the comparison, we either *reject* or *fail to reject* that the frequencies come from the same distribution.

Kolmogorov-Smirnov statistical test (K-S) compares two cumulative distribution functions (cdfs). For the ME and GC estimations, cdfs F_{GC} and F_{ME} can be compudet directly from pdfs f_{GC} and f_{ME}. Sampled cdf $F_{Sample}(x)$ is simply the number of samples $< x$, normalized by the total number of samples. Again, corresponding K-S test statistics are computed and compared to the critical value.

Last, we evaluate the precision of the 3 estimations by comparing true and estimated moments. For the purposes of this evaluation, we assume that we are given a set of higher moments T that have not been used to estimate the distributions. For each estimation, GC, ME, and sample-based, we compute the moments corresponding to the moments from T. The moments we compare were not used for ME/GC estimations and allow us to test how well the estimates generalize to capture higher moments as well. We compute *absolute* and *relative estimate errors*.

Experimental results suggest that moment-based GC and ME distribution estimations provide accurate approximations for continuous variables of probabilistic loops.

CHAPTER 7

Conclusion

In this thesis, we introduced a fully automated static analysis method for loop invariant generation for a class of probabilistic loops, pushing the limits of the state-of-the-art work. The invariants, MBIs, can capture moment-based properties of program variables of arbitrary order. We further used MBIs for the analysis of BNs.

The core method of this thesis was introduced in Chapter 3. The first key technique we utilized for the MBI generation is the removal of probabilities. We use statistical properties of distributions and moments to replace the former with the latter. Reasoning about moments rather than distributions makes the analysis feasible. Probabilistic updates of the loop are represented as (a larger number of) non-probabilistic updates over $\mathbb{E}$-variables.

Removal of probabilities is then combined with algebraic techniques from quantitative invariant generation from non-probabilistic programs to further rewrite the loop into linear recurrences over $\mathbb{E}$-variables. Closed-form solutions of the recurrences then give the moment-based invariant properties of the original loop.

We implemented the approach in a tool called MORA, described in detail in Chapter 4.

The theoretical understanding of the problem was further developed in Section 6.1, where we gave precise characterization for the class of moment-computable loops that can be analysed with our approach. The restrictions imposed on the loops are inherited from a non-probabilistic setting, and loosening these restrictions leads to undecidability. This shows that we have pushed to the edge of what is possible within the considered framework.

We also considered how the techniques of this thesis could be used when programs are not moment-computable. In Section 6.2, we looked into combinations of variables for which moment-based properties can be computed. In Section 6.3, we looked into approximating non-polynomial functions to get a moment-computable loop.

In the second part of the thesis, we applied moment-based reasoning about programs to the analysis of probabilistic models, in particular to Bayesian networks. We showed in Section 5.3

how to encode various classes of BNs, such as discrete, Gaussian, or conditional linear Gaussian BNs, as probabilistic programs. In Section 5.4, we also reformulated several challenges in BN analysis as the problem of computing MBIs, for instance, exact inference, sensitivity analysis, and filtering. This allowed us to address these challenges using the fully automated approach of MBI generation.

When an unbounded or continuous distribution is represented using moments, some information is necessarily lost. In Section 6.4, we explored how the moment-based loop properties/MBIs can be used to estimate the underlying distributions automatically, and we quantified how well the estimations recapture the original distribution.

In this thesis, we gave a strong theoretical foundation as well as an implementation of a fully automated moment-based analysis of probabilistic loops. It allows us to generate invariants over arbitrary higher-order moments for loops with polynomial updates and symbolic constants. We demonstrated that this approach can be used for probabilistic program analysis as well as the analysis of Bayesian networks.

List of Figures

List of Tables

List of Algorithms

Bibliography

[ABK+22] Daneshvar Amrollahi, Ezio Bartocci, George Kenison, Laura Kovács, Marcel Moosbrugger, and Miroslav Stankovic. Solving invariant generation for unsolvable loops. In *Static Analysis - 29th International Symposium, SAS 2022*, volume 13790 of *Lecture Notes in Computer Science*, pages 19–43. Springer, 2022. **Radhia Cousot Young Researcher Best Paper Award**.

[ACN17] Sheshansh Agrawal, Krishnendu Chatterjee, and Petr Novotnỳ. Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–32, 2017.

[AGR21] Alessandro Abate, Mirco Giacobbe, and Diptarko Roy. Learning probabilistic termination proofs. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, volume 12760 of *Lecture Notes in Computer Science*, pages 3–26. Springer, 2021.

[AMS20] Martin Avanzini, Georg Moser, and Michael Schaper. A modular cost analysis for probabilistic programs. *Proc. ACM Program. Lang.*, 4(OOPSLA):172:1–172:30, 2020.

[BCKR20] Jason Breck, John Cyphert, Zachary Kincaid, and Thomas Reps. Templates and recurrences: better together. *PLDI*, 2020.

[BEFH16] Gilles Barthe, Thomas Espitau, Luis María Ferrer Fioriti, and Justin Hsu. Synthesizing probabilistic invariants via Doob's decomposition. In *Proc. of CAV 2016: the 28th International Conference on Computer Aided Verification*, volume 9779 of *LNCS*, pages 43–61. Springer, 2016.

[BGB12] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Probabilistic relational hoare logics for computer-aided security proofs. In *MPC*, 2012.

[BGP+16] Olivier Bouissou, Eric Goubault, Sylvie Putot, Aleksandar Chakarov, and Sriram Sankaranarayanan. Uncertainty propagation using probabilistic affine forms and concentration of measure inequalities. In *TACAS*, 2016.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[BKK+23] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Lena Verscht. A calculus for amortized expected runtimes. *Proc. ACM Program. Lang.*, 7(POPL):1957–1986, 2023.

[BKKM18] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. How long, O Bayesian network, will I sample thee? - A program analysis perspective on expected sampling times. In *Proc. of ESOP 2018: the 27th European Symposium on Program. Languages and Systems*, volume 10801 of *LNCS*, pages 186–213. Springer, 2018.

[BKKV15] Tomás Brázdil, Stefan Kiefer, Antonín Kucera, and Ivana Hutarová Vareková. Runtime analysis of probabilistic programs with unbounded recursion. *Journal of Computer and System Sciences*, 2015.

[BKOB12] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *POPL*, 2012.

[BKS19] Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. Automatic generation of moment-based invariants for prob-solvable loops. In *Proc. of ATVA 2019: the 17th International Symposium on Automated Technology for Verification and Analysis*, volume 11781 of *LNCS*, pages 255–276. Springer, 2019.

[BKS20a] Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. Analysis of Bayesian networks via prob-solvable loops. In *Proc. of ICTAC 2020: the 17th International Colloquium on Theoretical Aspects of Computing*, volume 12545 of *LNCS*, pages 221–241. Springer, 2020.

[BKS20b] Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. Mora - automatic generation of moment-based invariants. In *Proc. of TACAS 2020: the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 12078 of *LNCS*, pages 492–498. Springer, 2020.

[BTP+22] Jialu Bao, Nitesh Trivedi, Drashti Pathak, Justin Hsu, and Subhajit Roy. Data-driven invariant learning for probabilistic programs. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*, volume 13371 of *Lecture Notes in Computer Science*, pages 33–54. Springer, 2022.

[CFGG20] Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. Polynomial invariant generation for non-deterministic recursive programs. In *PLDI*, 2020.

[CFN12] Anthony C. Constantinou, Norman E. Fenton, and Martin Neil. pi-football: A Bayesian network model for forecasting association football match outcomes. *Knowl. Based Syst.*, 36:322–339, 2012.

[CFNH16] Krishnendu Chatterjee, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. 2016.

[CGH+17] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of statistical software*, 76(1), 2017.

[CGMZ22] Krishnendu Chatterjee, Amir Kafshdar Goharshady, Tobias Meggendorfer, and Dorde Zikelic. Sound and complete certificates for quantitative termination analysis of probabilistic programs. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*, volume 13371 of *Lecture Notes in Computer Science*, pages 55–78. Springer, 2022.

[CH20] Jianhui Chen and Fei He. Proving almost-sure termination by omega-regular decomposition. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 869–882, 2020.

[CHWZ15] Yu-Fang Chen, Chih-Duo Hong, Bow-Yaw Wang, and Lijun Zhang. Counterexample-guided polynomial loop invariant generation by lagrange interpolation. In *Proc. of CAV 2015: the 27th International Conference on Computer Aided Verification*, volume 9206 of *LNCS*, pages 658–674. Springer, 2015.

[Coo90] Gregory F. Cooper. The computational complexity of probabilistic inference using Bayesian belief networks. *Artif. Intell.*, 42(2-3):393–405, 1990.

[CS13] Aleksandar Chakarov and Sriram Sankaranarayanan. Probabilistic program analysis with martingales. In *International Conference on Computer Aided Verification*, pages 511–526. Springer, 2013.

[CS14] Aleksandar Chakarov and Sriram Sankaranarayanan. Expectation Invariants for Probabilistic Program Loops as Fixed Points. *Static Analysis Symposium*, pages 85–100, 2014.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[DJKV17] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A storm is coming: A modern probabilistic model checker. In *Proc. of CAV 2017: the 29th International Conference on Computer Aided Verification*, volume 10427 of *LNCS*, pages 592–600. Springer, 2017.

[DL93] Paul Dagum and Michael Luby. Approximating probabilistic inference in Bayesian belief networks is NP-hard. *Artif. Intell.*, 60(1):141–153, 1993.

[dOBP16] Steven de Oliveira, Saddek Bensalem, and Virgile Prevosto. Polynomial invariants by linear algebra. In *ATVA*, 2016.

[Edw12] David Edwards. *Introduction to Graphical Modelling*. Springer Science & Business Media, 2012.

[FDG+19] Daniel J Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L Sangiovanni-Vincentelli, and Sanjit A Seshia. Scenic: a language for scenario specification and scene generation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 63–78, 2019.

[FH15] Luis María Ferrer Fioriti and Holger Hermanns. Probabilistic termination: Soundness, completeness, and compositionality. In *Proc. of POPL 2015: the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 489–501. ACM, 2015.

[FK15] Azadeh Farzan and Zachary Kincaid. Compositional recurrence analysis. pages 57–64. IEEE, 2015.

[FLNP00] Nir Friedman, Michal Linial, Iftach Nachman, and Dana Pe'er. Using Bayesian networks to analyze expression data. *J. Comput. Biol.*, 7(3-4):601–620, 2000.

[FLP85] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[FZJ+17] Yijun Feng, Lijun Zhang, David N. Jansen, Naijun Zhan, and Bican Xia. Finding polynomial loop invariants for probabilistic programs. In *Proc. of ATVA 2017: the 15th International Symposium on Automated Technology for Verification and Analysis*, volume 10482 of *LNCS*, pages 400–416. Springer, 2017.

[Gha15] Zoubin Ghahramani. Probabilistic machine learning and artificial intelligence. *Nat.*, 521(7553):452–459, 2015.

[GKM13] Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. Prinsys - on a quest for probabilistic loop invariants. In *Proc. of QEST 2013*, volume 8054 of *LNCS*, pages 193–208. Springer, 2013.

[GMV16] Timon Gehr, Sasa Misailovic, and Martin T. Vechev. PSI: Exact symbolic inference for probabilistic programs. In *CAV*, 2016.

[GS14] Noah D. Goodman and Andreas Stuhlmüller. The design and implementation of probabilistic programming languages. `http://dippl.org`, 2014. Accessed: 2022-9-26.

[Hec08] David Heckerman. A tutorial on learning with Bayesian networks. In *Innovations in Bayesian Networks: Theory and Applications*, volume 156 of *Studies in Computational Intelligence*, pages 33–82. Springer, 2008.

[Her90] Ted Herman. Probabilistic self-stabilization. *Inf. Process. Lett.*, 35(2):63–67, 1990.

[HJK17] Andreas Humenberger, Maximilian Jaroschek, and Laura Kovács. Automated generation of non-linear loop invariants utilizing hypergeometric sequences. *ISSAC*, 2017.

[HJK18a] Andreas Humenberger, Maximilian Jaroschek, and Laura Kovács. Aligator.jl - A Julia Package for Loop Invariant Generation. In *CICM*, pages 111–117, 2018.

[HJK18b] Andreas Humenberger, Maximilian Jaroschek, and Laura Kovács. Invariant generation for multi-path loops with polynomial assignments. In *VMCAI*, 2018.

[JC10] Xia Jiang and Gregory F. Cooper. A Bayesian spatio-temporal method for disease outbreak detection. *Journal of the American Medical Informatics Association*, 17(4):462–471, 2010.

[Kar94] Richard M. Karp. Probabilistic recurrence relations. *J. ACM*, 41(6):1136–1150, 1994.

[KBCR19] Zachary Kincaid, Jason Breck, John Cyphert, and Thomas W. Reps. Closed forms for numerical loops. *POPL*, 2019.

[KBS+13] Kenan Kalajdzic, Ezio Bartocci, Scott A. Smolka, Scott D. Stoller, and Radu Grosu. Runtime verification with particle filtering. In *Proc. of RV 2013, the 4th International Conference on Runtime Verification*, volume 8174 of *LNCS*, pages 149–166. Springer, 2013.

[KCBR17] Zachary Kincaid, John Cyphert, Jason Breck, and Thomas Reps. Non-linear reasoning for invariant synthesis. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–33, 2017.

[KF09] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models - Principles and Techniques*. MIT Press, 2009.

[KKM19] Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. On the hardness of analyzing probabilistic programs. *Acta Informatica*, 56(3):255–285, 2019.

[KKMO16] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. Weakest precondition reasoning for expected run–times of probabilistic programs. In *European Symposium on Programming*, pages 364–389. Springer, 2016.

[KMMM10] Joost-Pieter Katoen, Annabelle McIver, Larissa Meinicke, and Carroll C. Morgan. Linear-invariant generation for probabilistic programs: Automated support for proof-based methods. In *Proc. of SAS 2010*, volume 6337 of *LNCS*, pages 390–406, 2010.

[KMS+22a] Ahmad Karimi, Marcel Moosbrugger, Miroslav Stankovic, Laura Kovács, Ezio Bartocci, and Efstathia Bura. Distribution estimation for probabilistic loops. In *Proc. of QEST 2022: Quantitative Evaluation of Systems - 19th International Conference*, volume 13479 of *Lecture Notes in Computer Science*, pages 26–42. Springer, 2022.

[KMS+22b] Andrey Kofnov, Marcel Moosbrugger, Miroslav Stankovic, Ezio Bartocci, and Efstathia Bura. Moment-based invariants for probabilistic loops with non-polynomial assignments. In *Proc. of QEST 2022: Quantitative Evaluation of Systems - 19th International Conference*, volume 13479 of *Lecture Notes in Computer Science*, pages 3–25. Springer, 2022. **Best Paper Award**.

[KN10] Kevin B. Korb and Ann E. Nicholson. *Bayesian Artificial Intelligence*. Chapman and Hall, 2nd edition, 2010.

[KNP11] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. of CAV 2011*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[Kov08] Laura Kovács. Reasoning algebraically about P-solvable loops. In *TACAS*, pages 249–264, 2008.

[Koz81] Dexter Kozen. Semantics of probabilistic programs. *J. Comput. Syst. Sci.*, 22(3):328–350, 1981.

[Koz85] Dexter Kozen. A probabilistic PDL. *J. Comput. Syst. Sci.*, 30(2):162–178, 1985.

[KP11] Manuel Kauers and Peter Paule. *The Concrete Tetrahedron - Symbolic Sums, Recurrence Equations, Generating Functions, Asymptotic Estimates*. Springer, 2011.

[KUH19] Satoshi Kura, Natsuki Urabe, and Ichiro Hasuo. Tail probabilities for randomized program runtimes via martingales for higher moments. In *TACAS*, 2019.

[KZH+11] Joost-Pieter Katoen, Ivan S. Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David N. Jansen. The ins and outs of the probabilistic model checker MRMC. *Perform. Eval.*, 68(2):90–104, 2011.

[LG19] Ugo Dal Lago and Charles Grellois. Probabilistic termination by monadic affine sized typing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 41(2):1–65, 2019.

[Lin92] Gwo Dong Lin. *Characterizations of Distributions via Moments*, volume 54. Springer, 1992.

[LS88] S. L. Lauritzen and D. J. Spiegelhalter. Local Computation with Probabilities on Graphical Structures and their Application to Expert Systems (with discussion). *Royal Statistical Society: Series B (Statistical Methodology)*, 50(2):157–224, 1988.

[LSS05] Jiebo Luo, Andreas E. Savakis, and Amit Singhal. A Bayesian network-based framework for semantic image understanding. *Pattern Recognit.*, 38(6):919–934, 2005.

[MBKK21] Marcel Moosbrugger, Ezio Bartocci, Joost-Pieter Katoen, and Laura Kovács. Automated termination analysis of polynomial probabilistic programs. In *European Symposium on Programming*, pages 491–518. Springer, Cham, 2021.

[MHG21] Fabian Meyer, Marcel Hark, and Jürgen Giesl. Inferring expected runtimes of probabilistic integer programs using expected sizes. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part I*, volume 12651 of *Lecture Notes in Computer Science*, pages 250–269. Springer, 2021.

[MKB79] Kantilal Varichand Mardia, John T. Kent, and John M. Bibby. *Multivariate Analysis*. Academic Press, 1979.

[MM05] Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer, 2005.

[MMKK17] Annabelle McIver, Carroll Morgan, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. A new proof rule for almost-sure termination. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–28, 2017.

[Mod19] GeNIe Modeler. BayesFusion, LLC, 2019.

[MSBK22] Marcel Moosbrugger, Miroslav Stankovic, Ezio Bartocci, and Laura Kovács. This is the moment for probabilistic loops. *Proc. ACM Program. Lang.*, 6(OOPSLA2):1497–1525, 2022.

[MWG+18] Tom Minka, John M. Winn, John Guiver, Yordan Zaykov, Dany Fabian, and John Bronskill. Infer.NET 0.3, 2018. Microsoft Research Cambridge. http://dotnet.github.io/infer.

[NCH18] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. Bounded expectations: resource analysis for probabilistic programs. *ACM SIGPLAN Notices*, 53(4):496–512, 2018.

[NJ07] Richard E. Neapolitan and Xia Jiang. *Probabilistic Methods for Financial and Marketing Informatics*. Morgan Kaufmann, 2007.

[OKKM16] Federico Olmedo, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. Reasoning about recursive probabilistic programs. In *LICS*. ACM, 2016.

[Pea85] Judea Pearl. Bayesian networks: A model of self-activated memory for evidential reasoning. *Proceedings of the 7th Conference of the Cognitive Science Society*, pages 329–334, 1985.

[Pfe09] Avi Pfeffer. Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report*, 137:96, 2009.

[Rab80] Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of number theory*, 12(1):128–138, 1980.

[RCK04] Enric Rodríguez-Carbonell and Deepak Kapur. Automatic generation of polynomial loop invariants: Algebraic foundations. In *ISSAC*, 2004.

[RN10] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach*. Pearson Education, 2010.

[SBK22] Miroslav Stankovič, Ezio Bartocci, and Laura Kovács. Moment-based analysis of bayesian network properties. *Theoretical Computer Science*, 903:113–133, 2022.

[SRB+15] K. Selyunin, D. Ratasich, E. Bartocci, Md. A. Islam, S. A. Smolka, and R. Grosu. Neural programming: Towards adaptive control in cyber-physical systems. In *Proc. of CDC 2015*, pages 6978–6985. IEEE, 2015.

[WFG+19] Peixin Wang, Hongfei Fu, Amir Kafshdar Goharshady, Krishnendu Chatterjee, Xudong Qin, and Wenjun Shi. Cost analysis of nondeterministic probabilistic programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 204–220, 2019.

[WHR21] Di Wang, Jan Hoffmann, and Thomas Reps. Central moment analysis for cost accumulators in probabilistic programs. In *PLDI*, 2021.

[YD06] Changhe Yuan and Marek J. Druzdzel. Importance sampling algorithms for Bayesian networks: Principles and performance. *Mathematical and Computer Modelling*, 43(9):1189–1207, 2006.

[YS06] Håkan L. S. Younes and Reid G. Simmons. Statistical probabilistic model checking with a focus on time-bounded properties. *Inf. Comput.*, 2006.

[ZR98] Geoffrey Zweig and Stuart J. Russell. Speech recognition with dynamic Bayesian networks. In *Proc. of AAAI/IAAI 98, the 15th National Conference on Artificial Intelligence and 10th Innovative Applications of Artificial Intelligence Conference*, pages 173–180. AAAI Press / The MIT Press, 1998.