Modular Verification for Almost-Sure Termination of Probabilistic Programs[*]

MINGZHANG HUANG, BASICS Lab, Shanghai Jiao Tong University, China
HONGFEI FU[†], Shanghai Jiao Tong University, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China
KRISHNENDU CHATTERJEE, IST Austria (Institute of Science and Technology Austria), Austria
AMIR KAFSHDAR GOHARSHADY[‡], IST Austria (Institute of Science and Technology Austria), Austria

In this work, we consider the almost-sure termination problem for probabilistic programs that asks whether a given probabilistic program terminates with probability 1. Scalable approaches for program analysis often rely on modularity as their theoretical basis. In non-probabilistic programs, the classical variant rule (V-rule) of Floyd-Hoare logic provides the foundation for modular analysis. Extension of this rule to almost-sure termination of probabilistic programs is quite tricky, and a probabilistic variant was proposed by Fioriti and Hermanns in POPL 2015. While the proposed probabilistic variant cautiously addresses the key issue of integrability, we show that the proposed modular rule is still not sound for almost-sure termination of probabilistic programs.

Besides establishing unsoundness of the previous rule, our contributions are as follows: First, we present a sound modular rule for almost-sure termination of probabilistic programs. Our approach is based on a novel notion of descent supermartingales. Second, for algorithmic approaches, we consider descent supermartingales that are linear and show that they can be synthesized in polynomial time. Finally, we present experimental results on a variety of benchmarks and several natural examples that model various types of nested while loops in probabilistic programs and demonstrate that our approach is able to efficiently prove their almost-sure termination property.

CCS Concepts: • **Theory of computation** → **Logic and verification**; **Automated reasoning**; **Program verification**.

Additional Key Words and Phrases: Termination, Probabilistic Programs, Verification, Almost-Sure Termination

ACM Reference Format:
Mingzhang Huang, Hongfei Fu, Krishnendu Chatterjee, and Amir Kafshdar Goharshady. 2019. Modular Verification for Almost-Sure Termination of Probabilistic Programs. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 129 (October 2019), 29 pages. https://doi.org/10.1145/3360555

[*]A longer version, including appendices, is available at [Huang et al. 2019].
[†]Corresponding Author
[‡]Recipient of a DOC Fellowship of the Austrian Academy of Sciences (ÖAW)

Authors' addresses: Mingzhang Huang, BASICS Lab, Shanghai Jiao Tong University, Shanghai, China, mingzhanghuang@gmail.com; Hongfei Fu, Shanghai Jiao Tong University, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China, fuhf@cs.sjtu.edu.cn; Krishnendu Chatterjee, IST Austria (Institute of Science and Technology Austria), Klosterneuburg, Austria, krishnendu.chatterjee@ist.ac.at; Amir Kafshdar Goharshady, IST Austria (Institute of Science and Technology Austria), Klosterneuburg, Austria, amir.goharshady@ist.ac.at.

2475-1421/2019/10-ART129
https://doi.org/10.1145/3360555

1 INTRODUCTION

Probabilistic programs. Extending classical imperative programs with randomness, i.e. generation of random values according to probability distributions, gives rise to probabilistic programs [Gordon et al. 2014]. Such programs provide a flexible framework for many different applications, ranging from the analysis of network protocols [Foster et al. 2016; Kahn 2017; Smolka et al. 2017], to machine learning applications [Claret et al. 2013; Gordon et al. 2013; Roy et al. 2008; Ścibior et al. 2015], and robot planning [Thrun 2000, 2002]. The recent interest in probabilistic programs has led to many probabilistic programming languages (such as Church [Goodman et al. 2008], Anglican [Tolpin et al. 2016] and WebPPL [Goodman and Stuhlmüller 2014]) and their analysis is an active research area in formal methods and programming languages (see [Agrawal et al. 2018; Chakarov and Sankaranarayanan 2013; Chatterjee et al. 2016a,b; Esparza et al. 2012; Kaminski et al. 2016, 2018; Ngo et al. 2018; Wang et al. 2018]).

Termination problems. In program analysis, the most basic liveness problem is that of *termination*, that given a program asks whether it always terminates. In presence of probabilistic behavior, there are two natural extensions of the termination problem: first, the almost-sure termination problem that asks whether the program terminates with probability 1; and second, the finite termination problem that asks whether the expected termination time is finite. While finite termination implies almost-sure termination, the converse is not true. Both problems have been widely studied for probabilistic programs, e.g. [Chatterjee et al. 2016a,b; Kaminski et al. 2016, 2018].

Importance of Almost-Sure Termination. Almost-sure termination is the classical and most widely-studied problem that extends termination of non-probabilistic programs, and is considered as a core problem in the programming languages community. See [Agrawal et al. 2018; Chatterjee et al. 2016b, 2017; Fioriti and Hermanns 2015; McIver et al. 2018]. Proving finite termination of a program is much more ideal and probably the first goal of an analyzer. Indeed, another ideal scenario would be if we could prove sure termination, i.e. that every run of the program terminates. Unfortunately, in many real-world cases, sure or finite termination are either too hard to prove or do not hold for the interesting real-world programs in question. For example, consider Recursive Markov Chains (RMCs) and Stochastic Context-free Grammars (SCFGs) [Etessami and Yannakakis 2009], which are special cases of probabilistic programs and are widely used in the Natural Language Processing community [Etessami and Yannakakis 2009; Manning et al. 1999]. These programs are much simpler than general probabilistic programs, but finite termination does not hold for them, even in the real-world examples, even in the special cases such as RMCs with bounded number of entries and exits. This exact problem also appears in robot planning in AI, where many different types of random walks are known to be a.s. terminating but not finitely terminating. In such cases, there is a natural need for proving a.s. termination.

Modular approaches. Scalable approaches for program analysis are often based on modularity as their theoretical foundation. For non-probabilistic programs, the classical variant rule (V-rule) of Floyd-Hoare logic [Floyd 1967; Katz and Manna 1975] provides the necessary foundations for modular verification. Such modular methods allow decomposition of the programs into smaller parts, reasoning about the parts, and then combining the results to deduce the desired result for the entire program. Thus, they are the key technique in many automated methods for large programs.

Modular approaches for probabilistic programs. The modular approach for almost-sure termination of probabilistic programs was considered in [Fioriti and Hermanns 2015]. First, it was shown that a direct extension of the V-rule of non-probabilistic programs is not sound for almost-sure termination of probabilistic programs, as there is a crucial issue regarding integrability. Then, a modular rule, which cautiously addresses the integrability issue, was proposed as a sound rule for almost-sure termination of probabilistic programs. We refer to this rule as the FHV-rule.

Our contributions. Our main contributions are as follows:

(1) First, we show that the FHV-rule of [Fioriti and Hermanns 2015], which is the natural extension of the V-rule with integrability condition, is not sound for almost-sure termination of probabilistic programs. We do this by providing a concrete counterexample in which the FHV-rule deduces a.s. termination whereas the program is not a.s. terminating. We show that besides the issue of integrability, there is another crucial issue, regarding the non-negativity requirement in ranking supermartingales, that is not addressed by [Fioriti and Hermanns 2015] and leads to unsoundness.

(2) Second, we present a strengthened and sound modular rule for almost-sure termination of probabilistic programs that addresses both crucial issues. Our approach is based on a novel notion called "descent supermartingales" (DSMs), which is an important technical contribution of our work.

(3) Third, we provide a sound proof system for deducing a.s. termination of programs through DSMs. This proof system can be used by theorem provers to establish a.s. termination.

(4) Fourth, while we present our modular approach for general DSMs, for algorithmic approaches we focus on DSMs that are linear. We present an efficient polynomial-time algorithm for the synthesis of linear DSMs.

(5) Finally, we present an implementation of our synthesis algorithm for linear DSMs and demonstrate that our approach can handle benchmark programs used in [Ngo et al. 2018], is applicable to probabilistic programs containing various types of nested while loops, and can efficiently prove that these programs terminate almost-surely.

2 PRELIMINARIES

Throughout the paper, we denote by $\mathbb{N}$, $\mathbb{N}_0$, $\mathbb{Z}$, and $\mathbb{R}$ the sets of positive integers, non-negative integers, integers, and real numbers, respectively. We first review several useful concepts in probability theory and then present the syntax and semantics of our probabilistic programs.

2.1 Stochastic Processes and Martingales

We provide a short review of some necessary concepts in probability theory. For a more detailed treatment, see [Williams 1991].

Probability Distributions. A *discrete probability distribution* over a countable set U is a function $p : U \longrightarrow [0, 1]$ such that $\sum_{u \in U} p(u) = 1$. The *support* of p is defined as $\mathrm{supp}(p) := \{u \in U \mid p(u) > 0\}$.

Probability Spaces. A *probability space* is a triple $(\Omega, \mathcal{F}, \mathbb{P})$, where Ω is a non-empty set (called the *sample space*), $\mathcal{F}$ is a *σ-algebra* over Ω (i.e. a collection of subsets of Ω that contains the empty set $\emptyset$ and is closed under complementation and countable union) and $\mathbb{P}$ is a *probability measure* on $\mathcal{F}$, i.e. a function $\mathbb{P}\colon \mathcal{F} \longrightarrow [0, 1]$ such that (i) $\mathbb{P}(\Omega) = 1$ and (ii) for all set-sequences $A_1, A_2, \dots \in \mathcal{F}$ that are pairwise-disjoint (i.e. $A_i \cap A_j = \emptyset$ whenever $i \neq j$) it holds that $\sum_{i=1}^{\infty} \mathbb{P}(A_i) = \mathbb{P}\left(\bigcup_{i=1}^{\infty} A_i\right)$. Elements of $\mathcal{F}$ are called *events*. An event $A \in \mathcal{F}$ holds *almost-surely* (a.s.) if $\mathbb{P}(A) = 1$.

Random Variables. A *random variable* X from a probability space $(\Omega, \mathcal{F}, \mathbb{P})$ is an $\mathcal{F}$-measurable function $X\colon \Omega \longrightarrow \mathbb{R} \cup \{-\infty, +\infty\}$, i.e. a function such that for all $d \in \mathbb{R} \cup \{-\infty, +\infty\}$, the set $\{\omega \in \Omega \mid X(\omega) < d\}$ belongs to $\mathcal{F}$.

Expectation. The *expected value* of a random variable X from a probability space $(\Omega, \mathcal{F}, \mathbb{P})$, denoted by $\mathbb{E}(X)$, is defined as the Lebesgue integral of X w.r.t. $\mathbb{P}$, i.e. $\mathbb{E}(X) := \int X \, \mathrm{d}\mathbb{P}$. The precise definition of Lebesgue integral is somewhat technical and is omitted here (cf. [Williams 1991, Chapter 5] for a formal definition). If *range* $X = \{d_0, d_1, \dots\}$ is countable, then we have $\mathbb{E}(X) = \sum_{k=0}^{\infty} d_k \cdot \mathbb{P}(X = d_k)$.

Filtrations. A *filtration* of a probability space $(\Omega, \mathcal{F}, \mathbb{P})$ is an infinite sequence $\{\mathcal{F}_n\}_{n\in\mathbb{N}_0}$ of σ-algebras over Ω such that $\mathcal{F}_n \subseteq \mathcal{F}_{n+1} \subseteq \mathcal{F}$ for all $n \in \mathbb{N}_0$. Intuitively, a filtration models the information available at any given point of time.

Conditional Expectation. Let X be any random variable from a probability space $(\Omega, \mathcal{F}, \mathbb{P})$ such that $\mathbb{E}(|X|) < \infty$. Then, given any σ-algebra $\mathcal{G} \subseteq \mathcal{F}$, there exists a random variable (from $(\Omega, \mathcal{F}, \mathbb{P})$), denoted by $\mathbb{E}(X|\mathcal{G})$, such that:
(E1) $\mathbb{E}(X|\mathcal{G})$ is $\mathcal{G}$-measurable, and
(E2) $\mathbb{E}\left(|\mathbb{E}(X|\mathcal{G})|\right) < \infty$, and
(E3) for all $A \in \mathcal{G}$, we have $\int_A \mathbb{E}(X|\mathcal{G})\,\mathrm{d}\mathbb{P} = \int_A X\,\mathrm{d}\mathbb{P}$.
The random variable $\mathbb{E}(X|\mathcal{G})$ is called the *conditional expectation* of X given $\mathcal{G}$. The random variable $\mathbb{E}(X|\mathcal{G})$ is a.s. unique in the sense that if Y is another random variable satisfying (E1)–(E3), then $\mathbb{P}(Y = \mathbb{E}(X|\mathcal{G})) = 1$. We refer to [Williams 1991, Chapter 9] for details. Intuitively, $\mathbb{E}(X|\mathcal{G})$ is the expectation of X, when assuming the information in $\mathcal{G}$.

Discrete-Time Stochastic Processes. A *discrete-time stochastic process* is a sequence $\Gamma = \{X_n\}_{n\in\mathbb{N}_0}$ of random variables where X_n's are all from some probability space $(\Omega, \mathcal{F}, \mathbb{P})$. The process Γ is *adapted to* a filtration $\{\mathcal{F}_n\}_{n\in\mathbb{N}_0}$ if for all $n \in \mathbb{N}_0$, X_n is $\mathcal{F}_n$-measurable. Intuitively, the random variable X_i models some value at the i-th step of the process.

Difference-Boundedness. A discrete-time stochastic process $\Gamma = \{X_n\}_{n\in\mathbb{N}_0}$ adapted to a filtration $\{\mathcal{F}_n\}_{n\in\mathbb{N}_0}$ is *difference-bounded* if there exists a $c \in (0, \infty)$ such that for all $n \in \mathbb{N}_0$, $|X_{n+1} - X_n| \le c$ almost-surely.

Martingales and Supermartingales. A discrete-time stochastic process $\Gamma = \{X_n\}_{n\in\mathbb{N}_0}$ adapted to a filtration $\{\mathcal{F}_n\}_{n\in\mathbb{N}_0}$ is a *martingale* (resp. *supermartingale*) if for every $n \in \mathbb{N}_0$, $\mathbb{E}(|X_n|) < \infty$ and it holds a.s. that $\mathbb{E}(X_{n+1}|\mathcal{F}_n) = X_n$ (resp. $\mathbb{E}(X_{n+1}|\mathcal{F}_n) \le X_n$). We refer to [Williams 1991, Chapter 10] for a deeper treatment.

Intuitively, a martingale (resp. supermartingale) is a discrete-time stochastic process in which for an observer who has seen the values of $X_0, \ldots, X_n$, the expected value at the next step, i.e. $\mathbb{E}(X_{n+1}|\mathcal{F}_n)$, is equal to (resp. no more than) the last observed value X_n. Also, note that in a martingale, the observed values for $X_0, \ldots, X_{n-1}$ do not matter given that $\mathbb{E}(X_{n+1}|\mathcal{F}_n) = X_n$. In contrast, in a supermartingale, the only requirement is that $\mathbb{E}(X_{n+1}|\mathcal{F}_n) \le X_n$ and hence $\mathbb{E}(X_{n+1}|\mathcal{F}_n)$ may depend on $X_0, \ldots, X_{n-1}$. Also, note that $\mathcal{F}_n$ might contain more information than just the observations of X_i's.

Example 2.1. Consider an unbiased and discrete random walk, in which we start at a position X_0, and at each second walk one step to either left or right with equal probability. Let X_n denote our position after n seconds. It is easy to verify that $\mathbb{E}[X_{n+1}|X_0, \ldots, X_n] = \frac{1}{2}(X_n - 1) + \frac{1}{2}(X_n + 1) = X_n$. Hence, this random walk is a martingale. Note that by definition, every martingale is also a supermartingale. As another example, consider the classical gambler's ruin: a gambler starts with Y_0 dollars of money and bets continuously until he loses all of his money. If the bets are unfair, i.e. the expected value of his money after a bet is less than its expected value before the bet, then the sequence $\{Y_n\}_{n\in\mathbb{N}_0}$ is a supermartingale. In this case, Y_n is the gambler's total money after n bets. On the other hand, if the bets are fair, then $\{Y_n\}_{n\in\mathbb{N}_0}$ is a martingale.

Example 2.2 (Pólya's Urn [Mahmoud 2008]). As a more interesting example, consider an urn that initially contains R_0 red and B_0 blue marbles ($R_0 + B_0 > 0$). At each step, we take one marble from the urn, chosen uniformly at random, look at its color and then add two marbles of that color to the urn. Let B_n, R_n and M_n respectively be the number of red, blue and all marbles after n steps. Also, let $\beta_n = \frac{B_n}{M_n}$ and $\rho_n = \frac{R_n}{M_n}$ be the proportion of marbles that are blue (resp. red) after n steps. Let $\mathcal{F}_n$ model the observations until the n-th step. The process described above leads to the following equations:

$$M_{n+1} = 1 + M_n,$$
$$\mathbb{E}(B_{n+1}|\mathcal{F}_n) = \mathbb{E}(B_{n+1}|B_1, \ldots, B_n) = \frac{B_n}{M_n} \cdot (B_n + 1) + \frac{R_n}{M_n} \cdot B_n,$$
$$\mathbb{E}(R_{n+1}|\mathcal{F}_n) = \mathbb{E}(R_{n+1}|B_1, \ldots, B_n) = \frac{R_n}{M_n} \cdot (R_n + 1) + \frac{B_n}{M_n} \cdot R_n.$$

Note that we did not need to care about observing R_i's, M_i's, β_i's or ρ_i's, because they can be uniquely computed in terms of B_i's. More generally, an observer can observe only B_i's, or only R_i's, or only β_i's or ρ_i's and can then compute the rest using this information. Based on the equations above, we have:

$$\mathbb{E}(\beta_{n+1}|\mathcal{F}_n) = \frac{B_n}{M_n} \cdot \frac{B_n + 1}{M_n + 1} + \frac{M_n - B_n}{M_n} \cdot \frac{B_n}{M_n + 1} = \frac{B_n}{M_n} = \beta_n,$$
$$\mathbb{E}(\rho_{n+1}|\mathcal{F}_n) = \frac{R_n}{M_n} \cdot \frac{R_n + 1}{M_n + 1} + \frac{M_n - R_n}{M_n} \cdot \frac{R_n}{M_n + 1} = \frac{R_n}{M_n} = \rho_n.$$

Hence, both $\{\beta_n\}_{n \in \mathbb{N}_0}$ and $\{\rho_n\}_{n \in \mathbb{N}_0}$ are martingales. Informally, this means that the expected proportion of blue marbles in the next step is exactly equal to their observed proportion in the current step. This might be counter-intuitive. For example, consider a state where 99% of the marbles are blue. Then, it is more likely that we will add a blue marble in the next state. However, this is mitigated by the fact that adding a blue marble changes the proportions much less dramatically than adding a red marble.

2.2 Syntax

In the sequel, we fix two disjoint countable sets: the set of *program variables* and the set of *sampling variables*. Informally, program variables are directly related to the control flow of a program, while sampling variables represent random inputs sampled from distributions. We assume that every program variable is integer-valued, and every sampling variable is bound to a discrete probability distribution over integers. We first define several basic notions and then present the syntax.

Valuations. A *valuation* over a finite set V of variables is a function $\nu : V \longrightarrow \mathbb{Z}$ that assigns a value to each variable. The set of all valuations over V is denoted by Val_V.

Arithmetic Expressions. An *arithmetic expression* $\mathfrak{e}$ over a finite set V of variables is an expression built from the variables in V, integer constants, and arithmetic operations such as addition, multiplication, exponentiation, etc. For our theoretical results we consider a general setting for arithmetic expressions in which the set of allowed arithmetic operations can be chosen arbitrarily.

Propositional Arithmetic Predicates. A *propositional arithmetic predicate* over a finite set V of variables is a propositional formula ϕ built from (i) atomic formulae of the form $\mathfrak{e} \bowtie \mathfrak{e}'$ where $\mathfrak{e}, \mathfrak{e}'$ are arithmetic expressions and $\bowtie \in \{<, \leq, >, \geq\}$, and (ii) propositional connectives such as $\vee, \wedge, \neg$. The satisfaction relation $\models$ between a valuation ν and a propositional arithmetic predicate ϕ is defined through evaluation and standard semantics of propositional connectives, e.g. (i) $\nu \models \mathfrak{e} \bowtie \mathfrak{e}'$ iff $\mathfrak{e} \bowtie \mathfrak{e}'$ holds when the variables in $\mathfrak{e}, \mathfrak{e}'$ are substituted by their values in ν, (ii) $\nu \models \neg\phi$ iff $\nu \not\models \phi$ and (iii) $\nu \models \phi_1 \wedge \phi_2$ (resp. $\nu \models \phi_1 \vee \phi_2$) iff $\nu \models \phi_1$ and (resp. or) $\nu \models \phi_2$.

The Syntax. Our syntax is illustrated by the grammar in Figure 1. Below, we explain the grammar.

- *Variables.* Expressions $\langle pvar \rangle$ (resp. $\langle rvar \rangle$) range over program (resp. sampling) variables.
- *Arithmetic Expressions.* Expressions $\langle expr \rangle$ (resp. $\langle pexpr \rangle$) range over arithmetic expressions over all program and sampling variables (resp. all program variables).
- *Boolean Expressions.* Expressions $\langle bexpr \rangle$ range over propositional arithmetic predicates over program variables.

$$
\begin{array}{rcl}
\langle \mathit{prog} \rangle & ::= & \text{`}\mathbf{skip}\text{'} \\
& & \mid \langle \mathit{pvar} \rangle \text{ `:=' } \langle \mathit{expr} \rangle \\
& & \mid \langle \mathit{prog} \rangle \text{ `;' } \langle \mathit{prog} \rangle \\
& & \mid \text{`}\mathbf{if}\text{' } \langle \mathit{bexpr} \rangle \text{ `}\mathbf{then}\text{' } \langle \mathit{prog} \rangle \text{ `}\mathbf{else}\text{' } \langle \mathit{prog} \rangle \text{ `}\mathbf{fi}\text{'} \\
& & \mid \text{`}\mathbf{if}\text{' `}\star\text{' `}\mathbf{then}\text{' } \langle \mathit{prog} \rangle \text{ `}\mathbf{else}\text{' } \langle \mathit{prog} \rangle \text{ `}\mathbf{fi}\text{'} \\
& & \mid \text{`}\mathbf{if}\text{' `}\mathbf{prob}(p)\text{' `}\mathbf{then}\text{' } \langle \mathit{prog} \rangle \text{ `}\mathbf{else}\text{' } \langle \mathit{prog} \rangle \text{ `}\mathbf{fi}\text{'} \\
& & \mid \text{`}\mathbf{while}\text{' } \langle \mathit{bexpr} \rangle \text{ `}\mathbf{do}\text{' } \langle \mathit{prog} \rangle \text{ `}\mathbf{od}\text{'} \\
\\
\langle \mathit{literal} \rangle & ::= & \langle \mathit{pexpr} \rangle \text{ `}\leq\text{' } \langle \mathit{pexpr} \rangle \mid \langle \mathit{pexpr} \rangle \text{ `}\geq\text{' } \langle \mathit{pexpr} \rangle \\
\langle \mathit{bexpr} \rangle & ::= & \langle \mathit{literal} \rangle \mid \neg \langle \mathit{bexpr} \rangle \mid \langle \mathit{bexpr} \rangle \text{ `}\mathbf{or}\text{' } \langle \mathit{bexpr} \rangle \\
& & \mid \langle \mathit{bexpr} \rangle \text{ `}\mathbf{and}\text{' } \langle \mathit{bexpr} \rangle
\end{array}
$$

Fig. 1. The syntax of probabilistic programs

- *Programs.* A program can either be a single assignment statement (indicated by ':='), or '**skip**' which is the special statement that does nothing, or a conditional branch (indicated by '**if** ⟨*bexpr*⟩'), or a non-deterministic branch (indicated by '**if** $\star$'), or a probabilistic branch (indicated by '**if prob**(p)', where $p \in [0,1]$ is the probability of executing the **then** branch and $1-p$ that of the **else** branch), or a while loop (indicated by the keyword '**while**'), or a sequential composition of two subprograms (indicated by semicolon).

Program Counters. We assign a *program counter* to each assignment statement, skip, if branch and while loop. Intuitively, the counter specifies the current point in the execution of a program. We also refer to program counters as *labels.*

2.3 Semantics

To specify the semantics of our probabilistic programs, we follow previous approaches, such as [Chakarov and Sankaranarayanan 2013; Chatterjee et al. 2016a,b], and use Control Flow Graphs (CFGs) and Markov Decision Processes (MDPs) (see [Baier and Katoen 2008, Chapter 10]). Informally, a CFG describes how the program counter and valuations over program variables change along an execution of a program. Then, based on the CFG, one can construct an MDP as the semantical model of the probabilistic program.

Definition 2.3 (Control Flow Graphs). A *Control Flow Graph* (CFG) is a tuple

$$\mathcal{G} = (L, (V_\mathrm{p}, V_\mathrm{r}), \rightarrow) \tag{1}$$

with the following components:

- L is a finite set of *labels*, which is partitioned into the set L_b of *conditional branch* labels, the set L_a of *assignment* labels, the set L_p of *probabilistic* labels and the set L_d of *non-deterministic branch* labels;
- V_p and V_r are disjoint finite sets of *program* and *sampling* variables, respectively;
- $\rightarrow$ is a *transition relation* in which every member (called a *transition*) is a tuple of the form (ℓ, α, ℓ') for which (i) ℓ (resp. ℓ') is the *source label* (resp. *target label*) in L and (ii) α is either a propositional arithmetic predicate if $\ell \in L_\mathrm{b}$, or an *update function* $u : \mathit{Val}_{V_\mathrm{p}} \times \mathit{Val}_{V_\mathrm{r}} \rightarrow \mathit{Val}_{V_\mathrm{p}}$ if $\ell \in L_\mathrm{a}$, or $p \in [0,1]$ if $\ell \in L_\mathrm{p}$ or $\star$ if $\ell \in L_\mathrm{d}$.

We always specify an *initial* label $\ell_\mathrm{in} \in L$ representing the starting point of the program, and a *terminal* label $\ell_\mathrm{out} \in L$ that represents termination and has no outgoing transitions.

```
1:   while  $x \geq 1$  do
2:        $z := y$ ;
3:        while  $z \geq 0$  do
4:             if  $x < 2$  then
5:                  $x := x + r$
                 else
6:                   skip
                 fi ;
7:             $z := z - 1$
          od ;
8:        $y := 4 \times y$ ;
9:        $x := x - 1$
     od
10:
```

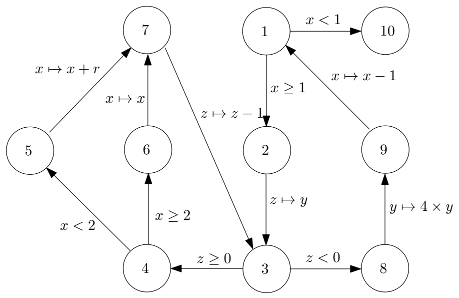

Fig. 2. A probabilistic program (left) and its CFG (right). In this program, $\mathbb{P}(r = 1) = \mathbb{P}(r = -1) = 0.5$.

Intuition for CFGs. Informally, a control flow graph specifies how the program counter and values for program variables change in a program. We have three types of labels, namely *branching*, *assignment* and *non-deterministic*. The initial label ℓ_{in} corresponds to the initial statement of the program. A conditional branch label corresponds to a conditional branching statement indicated by '**if** ϕ' or '**while** ϕ' , and leads to the next label determined by ϕ without changing the valuation. An assignment label corresponds to an assignment statement indicated by ':=' or **skip**, and leads to the next label right after the statement and an update to the value of the variable on the left hand side of ':=' that is specified by its right hand side. This update can be seen as a function that gives the next valuation over program variables based on the current valuation and the sampled values. The statement '**skip**' is treated as an assignment statement that does not change values. A probabilistic branch label corresponds to a probabilistic branching statement indicated by '**if prob**(p)', and leads to the label of '**then**' (resp. '**else**') branch with probability p (resp. $1 - p$). A non-deterministic branch label corresponds to non-deterministic choice statement indicated by '**if** $\star$', and has transitions to the two labels corresponding to the '**then**' and '**else**' branches.

By standard constructions, one can transform any probabilistic program into an equivalent CFG. We refer to [Chakarov and Sankaranarayanan 2013; Chatterjee et al. 2016a,b] for details.

Example 2.4. Consider the probabilistic program in Figure 2 (left). Its CFG is given in Figure 2 (right). In this program, x, y and z are program variables, and r is a sampling variable that observes the probability distribution $\mathbb{P}(r = 1) = \mathbb{P}(r = -1) = 0.5$. The numbers 1–10 are the program counters (labels). In particular, 1 is the initial label and 10 is the terminal label. The arcs represent transitions in the CFG. For example, the arc from 5 to 7 specifies the transition from label 5 to label 7 with the update function $x \mapsto x + r$ that assigns to program variable x, the value of the expression $x + r$, obtained by adding the value of x to a sampled value for the sampling variable r.

The Semantics. Based on CFGs, we define the semantics of probabilistic programs through the standard notion of Markov decision processes. Below, we fix a probabilistic program P with its CFG in form (1). We define the notion of *configurations* such that a configuration is a pair (ℓ, ν), where ℓ is a label (representing the current program counter) and $\nu \in Val_{V_{\text{p}}}$ is a valuation (representing the current valuation for program variables). We also fix a *sampling function* Υ which assigns to every sampling variable $r \in V_{\text{r}}$, a discrete probability distribution over $\mathbb{Z}$. Then, the *joint* discrete probability distribution $\overline{\Upsilon}$ over Val_{V_r} is defined as $\overline{\Upsilon}(\mu) := \prod_{r \in V_r} \Upsilon(r)(\mu(r))$ for all valuations μ over sampling variables.

The semantics is described by a Markov decision process (MDP). Intuitively, the MDP models the stochastic transitions, i.e. how the current configuration jumps to the next configuration. The state space of the MDP is the set of all configurations. The actions are τ, **th** and **el** and correspond to the absence of non-determinism, taking the **then**-branch of a non-deterministic branch label, and taking the **else**-branch of a non-deterministic branch label, respectively. The MDP transition probabilities are determined by the current configuration, the action chosen for the configuration and the statement at the current configuration.

To resolve non-determinism in MDPs, we use schedulers. A *scheduler* σ is a function which maps every history, i.e. all information up to the current execution point, to a probability distribution over the actions available at the current state. Informally, it resolves non-determinism by discrete probability distributions over actions that specify the probability of taking each action.

From the MDP semantics, the behavior of a probabilistic program P with its CFG in the form (1) is described as follows: Consider an arbitrary scheduler σ. The program starts in an initial configuration (ℓ_0, ν_0) where $\ell_0 = \ell_{\text{in}}$. Then in each step i ($i \geq 0$), given the current configuration (ℓ_i, ν_i), the next configuration (ℓ_{i+1}, ν_{i+1}) is determined as follows:

(1) a valuation μ_i of the sampling variables is sampled according to the joint distribution $\overline{\Upsilon}$;
(2) if $\ell_i \in L_a$ and (ℓ_i, u, ℓ') is the transition in $\to$ with source label ℓ_i and update function u, then (ℓ_{i+1}, ν_{i+1}) is set to be $(\ell', u(\nu_i, \mu_i))$;
(3) if $\ell_i \in L_b$ and $(\ell_i, \phi, \ell'), (\ell_i, \neg\phi, \ell'')$ are the two transitions in $\to$ with source label ℓ_i, then (ℓ_{i+1}, ν_{i+1}) is set to be either (i) (ℓ', ν_i) if $\nu_i \models \phi$, or (ii) (ℓ'', ν_i) if $\nu_i \models \neg\phi$;
(4) if $\ell_i \in L_d$ and $(\ell_i, \star, \ell'), (\ell_i, \star, \ell'')$ are the transitions in $\to$ with source label ℓ_i, then (ℓ_{i+1}, ν_{i+1}) is set to be (ℓ''', ν_i), where the label ℓ''' is chosen from ℓ', ℓ'' using the scheduler σ.
(5) if $\ell_i \in L_p$ and $(\ell_i, p, \ell'), (\ell_i, 1-p, \ell'')$ are the two transitions in $\to$ with source label ℓ_i, then (ℓ_{i+1}, ν_{i+1}) is set to be either (i) (ℓ', ν_i) with probability p, or (ii) (ℓ'', ν_i) with probability $1-p$;
(6) if there is no transition in $\to$ emitting from ℓ_i (i.e. if $\ell_i = \ell_{\text{out}}$), then (ℓ_{i+1}, ν_{i+1}) is set to be (ℓ_i, ν_i).

For a detailed construction of the MDP, see [Huang et al. 2019, Appendix A].

Runs and the Probability Space. A *run* is an infinite sequence of configurations. Informally, a run $\{(\ell_n, \nu_n)\}_{n \in \mathbb{N}_0}$ specifies that the configuration at the n-th step of a program execution is (ℓ_n, ν_n), i.e. the program counter (resp. the valuation for program variables) at the n-th step is ℓ_n (resp. ν_n). By construction, with an initial configuration $\mathbf{c}$ (as the initial state of the MDP) and a scheduler σ, the Markov decision process for a probabilistic program induces a unique probability space over the runs (see [Baier and Katoen 2008, Chapter 10] for details). In the rest of the paper, we denote by $\mathbb{P}^{\sigma}_{\mathbf{c}}$ the probability measure under the initial configuration $\mathbf{c}$ and the scheduler σ, and by $\mathbb{E}^{\sigma}_{\mathbf{c}}(-)$ the corresponding expectation operator.

3 PROBLEM STATEMENT

In this section, we define the modular verification problem of almost-sure termination over probabilistic programs. Below, we fix a probabilistic program P with its CFG in the form (1). We first define the notion of almost-sure termination. Informally, the property of almost-sure termination requires that a program terminates with probability 1. We follow the definitions in [Chakarov and Sankaranarayanan 2013; Chatterjee et al. 2016b; Fioriti and Hermanns 2015].

Definition 3.1 (Almost-sure Termination). A run $\omega = \{(\ell_n, \nu_n)\}_{n \in \mathbb{N}_0}$ of a program P is *terminating* if $\ell_n = \ell_{\text{out}}$ for some $n \in \mathbb{N}_0$. We define the *termination time* as a random variable T such that for a run $\omega = \{(\ell_n, \nu_n)\}_{n \in \mathbb{N}_0}$, $T(\omega)$ is the smallest n such that $\ell_n = \ell_{\text{out}}$ if such an n exists (this case corresponds to program termination), and ∞ otherwise (this corresponds to non-termination). The program P is said to be *almost-surely (a.s.) terminating* under initial configuration $\mathbf{c}$ if $\mathbb{P}^\sigma_\mathbf{c}(T < \infty) = 1$ for all schedulers σ.

Lemma 3.2. *Let the program P be the sequential (resp. branching) composition of two other programs P_1 and P_2, i.e. $P := P_1; P_2$ (resp. $P :=$ **if** $-$ **then** P_1 **else** P_2 **fi**), and assume that both P_1 and P_2 are a.s. terminating for any initial value. Then, P is also a.s. terminating for any initial value.*

Proof. We first prove the sequential case. Let $V_p = \{x_1, x_2, \ldots, x_m\}$ be the set of program variables in P and T, T_1, T_2 be the termination time random variables of P, P_1 and P_2, respectively. Define the vector F_1 of random variables as follows: if $\omega = \{(\ell_j, \nu_j)\}_{j\in\mathbb{N}_0}$ is a terminating run of P_1 with $T_1(\omega) = n$, i.e. if ω terminates at (ℓ_n, ν_n), then $F_1(\omega) = \nu_n$. Intuitively, $(F_1(\omega))_i$ is the random variable that models the value of the i-th program variable at termination time of P_1. Then, we have:

$$\mathbb{P}^\sigma_\mathbf{c}(T < \infty) = \sum_{\nu \in Val_{V_p}} \mathbb{P}^\sigma_\mathbf{c}(T_1 < \infty \ \wedge \ F_1 = \nu) \cdot \mathbb{P}^\sigma_\nu(T_2 < \infty).$$

Informally, P terminates if and only if P_1 terminates and then P_2, run with the initial valuation obtained from the last step of P_1, terminates as well. However, P_2 is a.s. terminating, hence $\mathbb{P}^\sigma_\nu(T_2 < \infty) = 1$ for all ν. Therefore,

$$\begin{aligned}\mathbb{P}^\sigma_\mathbf{c}(T < \infty) &= \sum_{\nu \in Val_{V_p}} \mathbb{P}^\sigma_\mathbf{c}(T_1 < \infty \ \wedge \ F_1 = \nu) \\ &= \mathbb{P}^\sigma_\mathbf{c}(T_1 < \infty) \\ &= 1,\end{aligned}$$

so P is also a.s. terminating.

For the branching case, note that if P does not terminate, then at least one of P_1 and P_2 does not terminate as well. Formally, $\mathbb{P}^\sigma_\mathbf{c}(T < \infty) \geq \mathbb{P}^\sigma_\mathbf{c}(T_1 < \infty) \cdot \mathbb{P}^\sigma_\mathbf{c}(T_2 < \infty) = 1$. □

Remark 1. *The lemma above shows that a.s. termination is closed under branching and sequential composition. Hence, in this paper, we focus on the major problem of modular verification for a.s. termination of nested while loops.*

Modular Verification. We now define the problem of modular verification of a.s. termination, following the terminology of [Kupferman and Vardi 1997]. We first describe the notion of modularity in general. Consider an operator op (e.g. sequential composition or loop nesting) over a set of objects. We say that a property ϕ is modular under the operator op with a *side condition* ψ (over pairs of objects) if we have

$$(\psi(O, O') \wedge \phi(O) \wedge \phi(O')) \Rightarrow \phi(\mathrm{op}(O, O')) \qquad (2)$$

holds for all objects O, O'. In other words, the modularity of the property ϕ says that if the side condition ψ and the property ϕ hold for O, O', then the property ϕ also holds on the (bigger) composed object $\mathrm{op}(O, O')$. The motivation behind modular verification is that it allows one to prove the property incrementally from sub-objects to the whole object.

The Almost-sure Termination Property. In this paper, we are concerned with a.s. termination of while loops. Our aim is to prove this based on the assumption that the loop body is a.s. terminating for *every* initial value. We consider $\textsc{Tm}(P)$ to be the target property expressing that the probabilistic program P is a.s. terminating for *every* initial value, and we consider the operator to be the while-loop operator **while**, i.e. given a probabilistic program P and a propositional arithmetic predicate G (as the loop guard), the probabilistic program **while**(G, P) is defined as **while** G **do** P **od**. Since

P might itself contain another while loop, our setting encompasses probabilistic nested loops of any depth.

We focus on the modular verification of $\textsc{Tm}(-)$ under the while-loop operator and solve the problem in the following steps. First, we establish a side condition ψ so that the assertion

$$(\psi(G, P) \wedge \textsc{Tm}(P)) \Rightarrow \textsc{Tm}(\textbf{while}(G, P)) \tag{3}$$

holds for all probabilistic programs P and propositional arithmetic predicates G[1]. Second, based on the proposed side condition, we explore possible proof-rule and algorithmic approaches.

Remark 2. *In modular verification, the level of modularity depends on the complexity of the side condition. The least amount of modularity is achieved when the side condition is equivalent to the target property, so that no information from sub-objects is used. On the other hand, maximum modularity is attained when there is no need to prove a side condition, resulting in what is sometimes called* compositionality *of the property*[2]*. In our modular approach to prove a.s. termination, we consider a side condition that lies in the middle: the side condition neither encodes the entire a.s. termination property, nor can it be removed. We note that it is not possible to find a modular approach for proving a.s. termination for nested while loops if we forbid side conditions on the loop guard and the loop body, because the variables in a loop guard can be updated in the inner loops.*

4 PREVIOUS APPROACHES FOR MODULAR VERIFICATION OF TERMINATION

In this section, we describe previous approaches for modular verification of the (a.s.) termination property for (probabilistic) while loops. We first present the variant rule from the Floyd-Hoare logic [Floyd 1967; Katz and Manna 1975] that is sound for non-probabilistic programs. Then, we describe the probabilistic extension proposed in [Fioriti and Hermanns 2015].

4.1 Classical Approach for Non-probabilistic Programs

Consider a non-probabilistic while loop

$$P = \textbf{while } G \textbf{ do } P_1; \ldots; P_n \textbf{ od}$$

where the programs $P_1, \ldots, P_n$ may contain nested while loops and are assumed to be terminating. The fundamental approach for modular verification is the following classical variant rule (V-rule) from the Floyd-Hoare logic [Floyd 1967; Katz and Manna 1975]:

$$\text{V-RULE}\ \frac{\begin{array}{c}\forall k : P_k \text{ terminates and } \{R = z\}P_k\{R \preceq z\} \\ \exists k\ \{R = z\}P_k\{R \prec z\}\end{array}}{\textbf{while } G \textbf{ do } P_1; \ldots; P_n \textbf{ od} \text{ terminates}}$$

In the V-rule above, R is an arithmetic expression over program variables that acts as a *ranking function*. The relation $\prec$ represents a well-founded relation when restricted to the loop guard G, while the relation $\preceq$ is the "non-strict" version of $\prec$ such that (i) $a \prec b \wedge b \preceq c \Rightarrow a \prec c$ and (ii) $a \preceq b \wedge b \prec c \Rightarrow a \prec c$. Then, the premise of the rule says that (i) for all P_k, the value of R after the execution of P_k does not increase in comparison with its initial value z before the execution, and (ii) there is some k such that the execution of P_k leads to a decrease in the value of R. If $\{R = z\}P_k\{R \preceq z\}$ holds, then P_k is said to be *unaffecting* for R. Similarly, if $\{R = z\}P_k\{R \prec z\}$ holds, then P_k is *ranking* for R. Informally, the variant rule says that if all P_k's are unaffecting and there is at least one P_k that is ranking, then P terminates.

[1]Note that we do not define or consider any assertion of the form $\textsc{Tm}(G)$, because checking the condition G always takes finite time.

[2]Some authors use the terms "modular" and "compositional" interchangeably, while others require compositional approaches to have no side conditions. To avoid confusion, we always describe our approach as modular.

The variant rule is sound for proving termination of non-probabilistic programs, because the value of R cannot be decremented infinitely many times, given that the relation $\prec$ is well-founded when restricted to the loop guard G.

4.2 A Previous Approach for Probabilistic Programs

The approach in [Fioriti and Hermanns 2015] can be viewed as an extension of the abstract V-rule, which is a proof system for a.s. terminating property. We call this abstract rule the FHV-rule:

$$\text{FHV-RULE}\ \frac{\begin{array}{c}\forall k : P_k \text{ terminates and } \{R = z\}P_k\{R \preceq z\} \\ \exists k\ \{R = z\}P_k\{R \prec z\}\end{array}}{\textbf{while } G \textbf{ do } P_1; \ldots ; P_n \textbf{ od} \ \text{ terminates}}$$

Note that while the FHV-rule looks identical to the V-rule, semantics of the Hoare triple in the FHV-rule are different from that of the V-rule (see below).

The FHV-rule is a direct probabilistic extension of the V-rule through the notion of *ranking supermartingales* (RSMs, see [Chakarov and Sankaranarayanan 2013; Chatterjee et al. 2016a,b]). RSMs are discrete-time stochastic processes that satisfy the following conditions: (i) their values are always non-negative; and (ii) at each step of the process, the conditional expectation of the value is decreased by at least a positive constant ϵ. The decreasing and non-negative nature of RSMs ensures that with probability 1 and in finite expected number of steps, the value of any RSM hits zero. When embedded into programs through the notion of RSM-maps (see e.g. [Chakarov and Sankaranarayanan 2013; Chatterjee et al. 2016b]), RSMs serve as a sound approach for proving termination of probabilistic programs with finite expected termination time, which implies a.s. termination, too.

In [Fioriti and Hermanns 2015], the R in the FHV-rule is a propositionally linear expression that represents an RSM, while $\prec$ is the well-founded relation on non-negative real numbers such that $x \prec y$ iff $x \leq y - \epsilon$ for some fixed positive constant ϵ and $\preceq$ is interpreted simply as $\leq$. Unaffecting and ranking conditions are extended to the probabilistic setting through conditional expectation (see $Dec_{\leq}(-,-), Dec_{<}(-,-)$ on [Fioriti and Hermanns 2015, Page 9]). Concretely, we say that (i) P_k is *unaffecting* if the expected value of R after the execution of P_k is no greater than its initial value before the execution; and (ii) P_k is *ranking* if the expected value of R after the execution of P_k is decreased by at least ϵ compared with its initial value before the execution. Note that in [Fioriti and Hermanns 2015], R is also called a *compositional RSM.*

Crucial Issue 1: Difference-boundedness and Integrability. In [Fioriti and Hermanns 2015], the authors accurately observed that simply extending the variant rule with expectation is not enough. They provided a counterexample in [Fioriti and Hermanns 2015, Section 7.2] that is not a.s. terminating but has a compositional RSM. The problem is that random variables may not be integrable after the execution of a probabilistic while loop. In order to resolve this integrability issue, they introduced the conditional difference-boundedness condition (see Section 2) for conditional expectation. Then, using the Optional Sampling/Stopping Theorem, they proved that, under this condition, the random variables are integrable after the execution of while loops. To ensure the conditional difference-bounded condition, they established sound inference rules (see [Fioriti and Hermanns 2015, Table 2 and Theorem 7.6]). With the integrability issue resolved, [Fioriti and Hermanns 2015] finally claims that compositional ranking supermartingales provide a sound approach for proving a.s. termination of probabilistic while loops (see [Fioriti and Hermanns 2015, Theorem 7.7]).

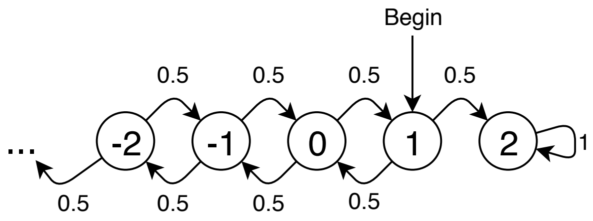

Fig. 3. The random walk modeled by the inner loop of the counterexample

5 A COUNTEREXAMPLE TO THE FHV-RULE

Although [Fioriti and Hermanns 2015] takes care of the integrability issue, we show that, unfortunately, the FHV-rule is still not sound. We present an explicit counterexample on which the FHV-rule proves a.s. termination, while the program is actually not a.s. terminating.

Example 5.1 (The Counterexample). Consider a 1-dimensional symmetric random walk as in Figure 3, in which at every location there is a $\frac{1}{2}$ probability of going left and $\frac{1}{2}$ probability of going right, except that there is a barrier at location 2, i.e. we remain at 2 if we reach it. Our inner loop in Figure 2 models this random walk, where the variable x corresponds to the location, and the number of steps taken is $z = y$ (lines $2 - 3$). In our outer loop, we run this random walk many times, but each time we increase the number of steps exponentially (line 8) and also, to avoid starting at the barrier, move the location one step to the left (line 9), before running the walk again.

Note that the program terminates only if $x \neq 2$, i.e. if we do *not* reach the barrier after some step-bounded random walk (lines 9 and 1). It is well-known that a 1-d symmetric random walk with a barrier will eventually reach the barrier with probability 1. In our program, the probability of reaching the barrier increases as we increase the number of steps of the random walk.

Moreover, at each iteration of the outer loop we are dramatically increasing the probability of reaching the barrier by increasing the number of steps by a factor of 4 (line 8). Therefore, after each iteration of the outer loop, the probability of termination in the next iteration is dramatically decreased and as a result, the program as a whole does not terminate a.s. This argument is formalized in Proposition 5.2.

We now argue that the FHV-rule wrongly deduces a.s. termination for this program. Intuitively, all the FHV-rule demands to assert termination is that (i) the inner loop terminates a.s., and (ii) there is an integrable expression R such that its expected value decreases in each iteration. Point (i) is trivial as the inner loop takes at most z steps. For point (ii), we let $R = x$, i.e. we use our location in the random walk as the ranking expression. It is easy to verify that the value of x does not change in expectation in the random walk. So, it decreases by 1 in each iteration of the outer loop (line 9). Hence, the FHV-rule incorrectly concludes that the program in Figure 2 terminates a.s. This is formalized in Proposition 5.3 (together with the argument for integrability). The flaw in the FHV-rule is that although the expected value of x decreases, at each iteration we reach $x = 2$ (the barrier) with higher and higher probability. So, despite the overall decrease in expectation, x reaches its maximum possible value with high probability, which leads to non-termination.

We now provide a rigorous proof of the above arguments.

Proposition 5.2. *The probabilistic program in Example 5.1 (Figure 2, Page 7) is not a.s. terminating. Specifically, it does not terminate with probability* 1 *when the initial value for the program variable x is* 1 *and the initial value for the program variable y is sufficiently large.*

Proof. The program does not terminate only if the value of x in label 9 is 2 after every execution of the inner loop. The key point is to prove that in the inner loop, the value of the program variable x will be 2 with higher and higher probability when the value of y increases. Consider the random walk in the inner loop. We abstract the values of x as three states '≤ 0', '1' and '2'. From the structure of the program, we have that if we start with the state '1', then after the inner loop, the successor state may transit to either '≤ 0', '1' or '2'. If the successor state is either '≤ 0' or '1', then the outer loop will terminate immediately. However, there is a positive probability that the successor state is '2' and the outer loop does not terminate in this loop iteration (as the value of x will be set back to 1). This probability depends on the steps of the random walk in the inner loop (determined by the value of y), and we show that it is higher and higher when the value of y increases. Thus, after more and more executions of the outer loop, the value of y continues to increase exponentially, and with higher and higher probability the program would be not terminating in the current execution of the loop body.

The detailed demonstration is as follows: W.l.o.g., we assume that $x = 1$ at every beginning of the inner loop. The values of x at label 9 are the results of the execution of inner loop with the same initial value, hence they are independent mutually. We now temporarily fix the value $\hat{y}$ for y at the beginning of the outer loop body and consider the probability that the value of x in label 9 is not 2. We use the random variable $\bar{X}_{\hat{y}}$ to describe the value of x at label 9 and analyze the situation $\bar{X}_{\hat{y}} \neq 2$ after the $\hat{y}$ iterations of the inner loop. Suppose that the $\hat{y}$ sampled values for r during the execution of the inner loop consist of m instances of -1 and $(\hat{y} - m)$ instances of 1. Since $\bar{X}_{\hat{y}} \neq 2$, we have $m \geq \frac{\hat{y}}{2}$. Then, there are $\binom{\hat{y}}{m} - \binom{\hat{y}}{m+1}$ different possible paths that avoid being absorbed by the barrier. The reason is that the only way to avoid absorption is to always have more -1's than 1's in any prefix of the path. Hence, the number of possible paths is the Catalan number. so we have $\mathbb{P}(\bar{X}_{\hat{y}} = 2) = 1 - \frac{1}{2^{\hat{y}}} \sum_{\frac{\hat{y}}{2} \leq m \leq \hat{y}} (\binom{\hat{y}}{m} - \binom{\hat{y}}{m+1}) = 1 - \frac{1}{2^{\hat{y}}} \binom{\hat{y}}{\lceil \frac{\hat{y}}{2} \rceil}$. Since $\sqrt{2\pi} n^{n+\frac{1}{2}} e^{-n} \leq n! \leq e n^{n+\frac{1}{2}} e^{-n}$ for $n \geq 1$ (applying Stirling's approximation), we have $1 - \frac{1}{2^{\hat{y}}} \binom{\hat{y}}{\lceil \frac{\hat{y}}{2} \rceil} = 1 - \frac{\hat{y}!}{2^{\hat{y}} (\frac{\hat{y}}{2}!)^2} \geq 1 - \frac{e \hat{y}^{\hat{y}+\frac{1}{2}} e^{-\hat{y}}}{2^{\hat{y}} (\sqrt{2\pi} \frac{\hat{y}}{2}^{\frac{\hat{y}}{2}+\frac{1}{2}} e^{-\frac{\hat{y}}{2}})^2} = 1 - \frac{e}{\pi \sqrt{\hat{y}}}$ for every even $\hat{y}$. Note that $\mathbb{P}(T = \infty) = \prod_{i \in \mathbb{N}_0} \mathbb{P}(\bar{X}_{\hat{y}_i} = 2)$, where $\hat{y}_i$ is the value of y at the i-th arrival to the label 9 and recall that $\hat{y}_0$ is sufficiently large. Furthermore, from the program we have $\hat{y}_i = 4^i \cdot \hat{y}_0$. Let $d := \frac{e}{\pi \sqrt{\hat{y}_0}}$, and we obtain that $\mathbb{P}(T = \infty) = \prod_{i \in \mathbb{N}_0} \mathbb{P}(\bar{X}_{\hat{y}_i} = 2) = \prod_{i \in \mathbb{N}_0} (1 - \frac{1}{2^{\hat{y}_i}} \binom{\hat{y}_i}{\lceil \frac{\hat{y}_i}{2} \rceil}) \geq \prod_{i \in \mathbb{N}_0} (1 - \frac{d}{\sqrt{4^i}})$. A well-known convergence criterion for infinite products is that $\prod_{i \in \mathbb{N}_0} (1 - q_i)$ converges to a non-zero number if and only if $\sum_{i \in \mathbb{N}_0} q_i$ converges for $0 \leq q_i < 1$. Since $\sum_{i \in \mathbb{N}_0} \frac{d}{2^i}$ converges to $2d$, we have the infinite product $\prod_{i \in \mathbb{N}_0} (1 - \frac{d}{2^i})$ converges to a non-zero number. Thus, $\mathbb{P}(T = \infty) > 0$. □

We now show that, using the FHV-rule proposed in [Fioriti and Hermanns 2015], one can deduce that the probabilistic program in Example 2 is a.s. terminating.

Proposition 5.3. *The FHV-rule in [Fioriti and Hermanns 2015] derives that the probabilistic program in Example 2 is a.s. terminating.*

Proof. To see that the FHV-rule derives a.s. termination on this example, we show that the expression x is a compositional RSM that satisfies the integrability and difference-boundedness conditions. First, we can show that the program variable x is integrable and difference-bounded at every label. For example, for assignment statements at labels 2, 5, 7, 8 and 9 in Figure 2, the expression x is integrable and difference-bounded after these statements simply because either they do not involve x at the left hand side or the assignment changes the value of x by 1 unit. Similarly, within the nested loop, the loop body (from label 4 to label 7) causes bounded change to the value of x, so the expression x is integrable after the inner loop (using the while-rule in [Fioriti and Hermanns 2015, Table 2]). Second, it is easy to see that the expression x is a compositional RSM as from [Fioriti and Hermanns 2015, Definition 7.1] we have the following:

- The value of x does not increase after the assignment statements $z := y$ and $y := 4 \times y$;
- In the loop body of the nested loop, the expected value of x does not increase, given that it does not increase in any of the conditional branches;
- By definition of $Dec_{\leq}(-,-)$, the expected value of x does not increase after the inner loop;
- The value of x is decreased by 1 after the last assignment statement $x := x - 1$.

Thus, by applying [Fioriti and Hermanns 2015]'s main theorem for compositionality ([Fioriti and Hermanns 2015, Theorem 7.7]), we can conclude that the program should be a.s. terminating. □

From Proposition 5.2 and Proposition 5.3, we establish the main theorem of this section, i.e. that the FHV-rule is not sound.

Theorem 5.4. *The FHV-rule, i.e. the probabilistic extension of the V-rule as proposed in [Fioriti and Hermanns 2015], is not sound for a.s. termination of probabilistic programs, even if we require the compositional RSM R to be difference-bounded and integrable.*

Note that integrability is a very natural requirement in probability theory. Hence, Theorem 5.4 states that a natural probabilistic extension of the variant rule is not sufficient for proving a.s. termination of probabilistic programs.

Crucial Issue 2: Non-negativity of RSMs. The reason why the approach of [Fioriti and Hermanns 2015] is not sound lies in the fact that their approach neglects the non-negativity of ranking supermartingales (RSMs), as their compositional RSMs are not required to be non-negative. In the classical V-rule for non-probabilistic programs, non-negativity is not required, given that negative values in a non-probabilistic setting simply mean that R is negative. However, in the presence of probability, the expected value of R is taken into account, and not R itself. Thus, it is possible that the expected value of R decreases and becomes arbitrarily negative, tending to $-\infty$, while simultaneously the value of R increases with higher and higher probability. In our counterexample (Example 5.1), the expected value of x decreases after each outer loop iteration, however the probability that the value of x remains the same increases with the value of y. More specifically, the decrease in the expected value results from the fact that after the inner loop, the value of x may get arbitrarily negative towards $-\infty$. For a detailed explanation of the unsoundness of the FHV-rule, see [Huang et al. 2019, Appendix B].

6 OUR MODULAR APPROACH

In the previous section, we showed that the FHV-rule is not sound for proving a.s. termination of probabilistic programs. In this section, we show how the FHV-rule can be minimally strengthened to a sound approach. The general idea of our approach is to define a new notion called a *Descent Supermartingale Map (DSM)* that requires the expected value of the expression R in the variant rule to always decrease by at least a positive amount ϵ. We call this the *strict decrease* condition. This condition is in contrast with the FHV-rule that allows the value of R at certain statements to remain the same (in expectation). We show that after this strengthening, the resulting rule is sound for modular verification of a.s. termination over probabilistic programs. We handle Crucial Issue 1 in the same manner as in the FHV-rule, i.e. by enforcing difference-boundedness. As for Crucial Issue 2, we show that unlike RSMs, DSMs do not require non-negativity. Hence, this issue does not apply to our approach. Our main mathematical tools are the *concentration inequalities* (e.g. [McDiarmid 1998]) that give tight upper bounds on the probability that a stochastic process deviates from its mean value.

In this section, we present our approach in terms of martingales and prove its soundness. In the next sections, we provide both a proof system based on inference rules and a synthesis algorithm based on templates for obtaining DSM-maps. This shows that our approach (i) leads to a completely automated method for proving a.s. termination (similar to other martingale-based approaches such as [Chakarov and Sankaranarayanan 2013; Chatterjee et al. 2016a,b]) and (ii) can also be applied in a semi-automatic setting using interactive theorem provers (similar to other rule-based approaches such as [Kaminski et al. 2016; McIver et al. 2018; Olmedo et al. 2016]). Hence, our approach combines the best aspects of both martingale-based and rule-based approaches, with the added benefit of being modular.

To clarify that our approach is indeed a strengthening of the FHV-rule in [Fioriti and Hermanns 2015], we first write the rule-based approach of [Fioriti and Hermanns 2015] in an equivalent martingale-based format. Below, we fix a probabilistic program P' and a loop guard G and let $P := \textbf{while}(G, P')$. For the purpose of modular verification, we assume that P' is a.s. terminating. We recall that T is the termination-time random variable (see Definition 3.1) and $\overline{\Upsilon}$ is the joint discrete probability distribution for sampling variables. We also use the standard notion of invariants, which are over-approximations of the set of reachable configurations at every label.

Invariants. An *invariant* is a function $I : L \longrightarrow 2^{Val_{V_P}}$, such that for each label $\ell \in L$, the set $I(\ell)$ at least contains all valuations v of program variables for which the configuration (ℓ, v) can be visited in some run of the program. An invariant I is *linear* if every $I(\ell)$ is a finite union of polyhedra.

We can now describe the FHV-rule approach in [Fioriti and Hermanns 2015] using *V-rule supermartingale maps.* A *V-rule supermartingale map* w.r.t. an invariant I is a function $R : Val_{V_\mathrm{p}} \longrightarrow \mathbb{R}$ satisfying the following conditions:

- *Non-increasing property.* The value of R does not increase in expectation after the execution of any of the statements in the outer-loop body. For example, the non-increasing condition for an assignment statement $\ell \in L_\mathrm{a}$ with $(\ell, u, \ell') \in \longrightarrow$ (recall that u is the update function) is equivalent to $\sum_{\mu \in Val_{V_\mathrm{r}}} \overline{\Upsilon}(\mu) \cdot R(u(v, \mu)) \leq R(v)$ for all $v \in I(\ell)$. This condition can be similarly derived for other types of labels.
- *Decrease property.* There exists a statement that will definitely be executed in every loop iteration and will cause R to decrease (in expectation). For example, the condition for strict decrease at an assignment statement $\ell \in L_\mathrm{a}$ with $(\ell, u, \ell') \in \longrightarrow$ says that for all $v \in I(\ell)$ we have $\sum_{\mu \in Val_{V_\mathrm{r}}} \overline{\Upsilon}(\mu) \cdot R(u(v, \mu)) \leq R(v) - \epsilon$, where ϵ is a fixed positive constant.
- *Well-foundedness.* The values of R should be bounded from below when restricted to the loop guard. Formally, this condition requires that for a fixed constant c and all $v \in I(\ell_\mathrm{in})$ such that $v \models G$, we have $R(v) \geq c$.
- *Conditional difference-boundedness.* The conditional expected change in the value of R after the execution of each statement is bounded. For example, at an assignment statement $\ell \in L_\mathrm{a}$ with $(\ell, u, \ell') \in \longrightarrow$, this condition says that there exists a fixed positive bound d, such that $\sum_{\mu \in Val_{V_\mathrm{r}}} \overline{\Upsilon}(\mu) \cdot |R(u(v, \mu)) - R(v)| \leq d$ for all $v \in I(\ell)$. The purpose of this condition is to ensure the integrability of R (see [Fioriti and Hermanns 2015, Lemma 7.4]).

Strengthening. We strengthen the FHV-rule of [Fioriti and Hermanns 2015] in two ways. First, as the major strengthening, we require that the expression R should strictly decrease in expectation at every statement, as opposed to [Fioriti and Hermanns 2015] where the value of R is only required to decrease at some statement. Second, we slightly extend the conditional difference-boundedness condition and require that the difference caused in the value of R after the execution of each statement should always be bounded, i.e. we require difference-boundedness not only in expectation, but in every run of the program.

The core notion in our strengthened approach is that of *Descent Supermartingale maps (DSM-maps).* A DSM-map is a function representing a decreasing amount (in expectation) at each step of the execution of the program.

Definition 6.1 (Descent Supermartingale Maps). A *descent supermartingale map* (DSM-map) w.r.t. real numbers $\epsilon > 0$, $c \in \mathbb{R}$, a non-empty interval $[a, b] \subseteq \mathbb{R}$ and an invariant I is a function $\eta : L \times Val_{V_p} \to \mathbb{R}$ satisfying the following conditions:

(D1) For each $\ell \in L_a$ with $(\ell, u, \ell') \in \to$, it holds that
- $a \le \eta(\ell', u(v, \mu)) - \eta(\ell, v) \le b$ for all $v \in I(\ell)$ and $\mu \in Val_{V_r}$;
- $\sum_{\mu \in Val_{V_r}} \overline{\Upsilon}(\mu) \cdot \eta(\ell', u(v, \mu)) \le \eta(\ell, v) - \epsilon$ for all $v \in I(\ell)$;

(D2) For each $\ell \in L_b$ and $(\ell, \phi, \ell') \in \to$, it holds that $a \le \eta(\ell', v) - \eta(\ell, v) \le \min\{-\epsilon, b\}$ for all $v \in I(\ell)$ such that $v \models \phi$;

(D3) For each $\ell \in L_d$ and $(\ell, \star, \ell') \in \to$, it holds that $a \le \eta(\ell', v) - \eta(\ell, v) \le \min\{-\epsilon, b\}$ for all $v \in I(\ell)$;

(D4) For each $\ell \in L_p$ with $(\ell, p, \ell'), (\ell, 1 - p, \ell'') \in \to$, it holds that
- $a \le \eta(\ell', v) - \eta(\ell, v) \le b$ for all $v \in I(\ell)$,
- $a \le \eta(\ell'', v) - \eta(\ell, v) \le b$ for all $v \in I(\ell)$,
- $p \cdot \eta(\ell', v) + (1 - p) \cdot \eta(\ell'', v) \le \eta(\ell, v) - \epsilon$ for all $v \in I(\ell)$;

(D5) For all $v \in I(\ell_{in})$ such that $v \models G$ (recall that G is the loop guard), it holds that $\eta(\ell_{in}, v) \ge c$.

Informally, R is a DSM-map if:

(D1)–(D4) Its value decreases in expectation by at least ϵ after the execution of each statement (the strict decrease condition), and its change of value before and after each statement falls in $[a, b]$ (the strengthened difference-boundedness condition);

(D5) Its value is bounded from below by c at every entry into the loop body (the well-foundedness condition).

Remark 3. *We remark two points about DSM-maps:*
- *DSM-maps require well-foundedness only w.r.t. the outermost loop guard. See (D5) above.*
- *The function η is dependent not only on the valuation, but also on the label (program counter). Hence, it can correspond to different expressions at each label. Informally, we do not have a single fixed expression R, but instead have a label-dependent expression $\eta(\ell)$. This gives our approach more flexibility.*

By the decreasing nature of DSM-maps, it is intuitively true that the existence of a DSM-map implies a.s. termination. However, this point is non-trivial as counterexamples will arise if we drop the difference-boundedness condition and only require the strict decrease condition (see e.g. [Huang et al. 2018, Example 3]). In the following, we use the difference-boundedness condition to derive a concentration property on the termination time (see [Chatterjee et al. 2016b]). Under this concentration property, we prove that DSM-maps are sound for proving a.s. termination.

We first present a well-known concentration inequality called *Hoeffding's Inequality.*

Theorem (Hoeffding's Inequality [Hoeffding 1963]). *Let $\{X_n\}_{n \in \mathbb{N}_0}$ be a supermartingale w.r.t. some filtration $\{\mathcal{F}_n\}_{n \in \mathbb{N}}$ and $\{[a_n, b_n]\}_{n \in \mathbb{N}}$ be a sequence of intervals with positive length in $\mathbb{R}$. If X_0 is a constant random variable and $X_{n+1} - X_n \in [a_n, b_n]$ a.s. for all $n \in \mathbb{N}_0$, then*

$$\mathbb{P}(X_n - X_0 \ge \lambda) \le \exp(-\frac{2\lambda^2}{\sum_{k=1}^{n}(b_k - a_k)^2})$$

for all $n \in \mathbb{N}_0$ and $\lambda > 0$.

Hoeffding's Inequality states that for any difference-bounded supermartingale, it is unlikely that its value X_n at the n-th step exceeds its initial value X_0 by much (measured by λ).

Using Hoeffding's Inequality, we prove the following lemma.

Lemma 6.2. *Let $\{X_n\}_{n\in\mathbb{N}_0}$ be a supermartingale w.r.t. some filtration $\{\mathcal{F}_n\}_{n\in\mathbb{N}}$ and $[a,b]$ be an interval with positive length in $\mathbb{R}$. If X_0 is a constant random variable, it holds that $\mathbb{E}(X_{n+1}|\mathcal{F}_n) \leq X_n - \epsilon$ for some $\epsilon > 0$ and $X_{n+1} - X_n \in [a,b]$ a.s. for all $n \in \mathbb{N}_0$, then for any $\lambda \in \mathbb{R}$,*

$$\mathbb{P}(X_n - X_0 \geq \lambda) \leq \exp(-\frac{2(\lambda + n\cdot\epsilon)^2}{n(b-a)^2})$$

for all sufficiently large n.

Proof. Let $Y_n = X_n + n\cdot\epsilon$, then $a + \epsilon \leq Y_{n+1} - Y_n = X_{n+1} - X_n + \epsilon \leq b + \epsilon$. Given that

$$\begin{aligned}\mathbb{E}(Y_{n+1}|\mathcal{F}_n) &= \mathbb{E}(X_{n+1}|\mathcal{F}_n) + (n+1)\cdot\epsilon \\ &\leq X_n + n\cdot\epsilon \\ &= Y_n,\end{aligned}$$

we conclude that $\{Y_n\}_{n\in\mathbb{N}_0}$ is a supermartingale. Now we apply Hoeffding's Inequality for all n such that $\lambda + n\cdot\epsilon > 0$, and we get

$$\begin{aligned}\mathbb{P}(X_n - X_0 \geq \lambda) &= \mathbb{P}(Y_n - Y_0 \geq \lambda + n\cdot\epsilon) \\ &\leq \exp(-\frac{2(\lambda + n\cdot\epsilon)^2}{n(b-a)^2})\end{aligned}$$

□

Thus, we have the following corollary by calculation.

Corollary 6.3. *Let $\{X_n\}_{n\in\mathbb{N}_0}$ be a supermartingale satisfying the conditions of Lemma 6.2. Then,* $\lim_{n\to+\infty}\sum_{k=n}^{+\infty}\mathbb{P}(X_k - X_0 \geq \lambda) = 0$.

We are now ready to prove the soundness of DSM-maps.

Theorem 6.4 (Soundness of DSM-maps). *Let $P = \textbf{while}(G, P')$. If (i) P' terminates a.s. for any initial valuation and scheduler; and (ii) there exists a DSM-map η for P, then for any initial valuation $v^* \in Val_{V_p}$ and for all schedulers σ, we have $\mathbb{P}^{\sigma}_{v^*}(T < \infty) = 1$.*

Proof Sketch. Let ϵ, c, a, b be as defined in Definition 6.1. For a given program P with its DSM-map η, we define the stochastic process $\{X_n = \eta(\ell_n, v_n)\}_{n\in\mathbb{N}_0}$ where (ℓ_n, v_n) is the pair of random variables that represents the configuration at the n-th step of a run. We also define the stochastic process $\{B_n\}_{n\in\mathbb{N}_0}$ in which each B_n represents the number of steps in the execution of P until the n-th arrival at the initial label ℓ_{in}. Then, X_{B_n} is the random variable representing the value of η at the n-th arrival at ℓ_{in}. Recall that, by condition (D5) in the definition of DSM-maps, the program stops if $X_{B_n} < c$. We now prove the crucial property that $\mathbb{P}(T' < \infty) \geq 1 - \lim_{n\to\infty}\mathbb{P}(X_{B_n} \geq c) = 1$, where T' is the random variable that measures the number of outer loop iterations in a run. We want to estimate the probability of $\mathbb{P}(X_{B_n} \geq c)$ which is bounded by $\sum_{k=n}^{+\infty}\mathbb{P}(X_k \geq c)$. Note that X_n satisfies the conditions of Lemma 6.2. We use Corollary 6.3 to bound the probability. Since $\mathbb{P}(T < \infty) = 1$ iff $\mathbb{P}(T' < \infty) = 1$ (as P' is a.s. terminating), we obtain that $\mathbb{P}(T < \infty) = 1$. For a more detailed proof, see [Huang et al. 2019, Appendix C].

□

Remark 4 (Modularity). *The theorem above directly leads to a modular approach for proving a.s. termination. To prove that P terminates a.s., it suffices to first prove that P' terminates a.s., and then show the existence of a DSM-map w.r.t. P as the side condition. We stress that in the theorem*

```
1 :  while $x \geq 1$ do
2 :    $y := r$;
3 :    while $y \geq 1$ do
4 :        if $\star$ then
5 :            if prob (6/13) then
6 :                $x := x + 1$
                 else
7 :                $x := x - 1$
                 fi
             else
8 :            if prob (4/13) then
9 :                $x := x + 2$
                 else
10 :               $x := x - 1$
                 fi
             fi ;
11 :       $y := y - 1$
         od
      od
12 :
```

Fig. 4. An Example Probabilistic Program. In this program, $\mathbb{P}(r = k) = 1/9$ for $k = 1, 2, \ldots, 9$.

above, it is necessary to assume that P' terminates a.s., because DSM-maps consider well-foundedness and termination only w.r.t. the outermost loop guard (see Remark 3). Hence, the existence of a DSM-map does not prove a.s. termination in and of itself, but only serves as a side condition in our modular approach.

We illustrate an example application of Theorem 6.4.

Example 6.5. Consider the probabilistic while loop in Figure 4. where the probability distribution for the sampling variable r is given by $\mathbb{P}(r = k) = 1/9$ for $k = 1, 2, \ldots, 9$.

The while loop models a variant of gambler's ruin based on the mini-roulette game with 13 slots [Chatterjee et al. 2018]. Initially, the gambler has x units of money and he continues betting until he has no money. At the start of each outer loop iteration, the number of gambling rounds is chosen uniformly at random from $1, 2, \ldots, 9$ (i.e. the program variable y is the number of gambling rounds in this iteration). Then, at each round, the gambler takes one unit of money, and either chooses an *even-money bet* that bets the ball to stop at even numbers between 1 and 13, which has a probability of $\frac{6}{13}$ to win one unit of money (see the non-deterministic branch from label 5 to label 7), or a 2-*to*-1 *bet* that bets the ball to stop at 4 selected slots and wins two units of money with probability $\frac{4}{13}$ (see the branch from label 8 to label 10). During each outer loop iteration, it is possible that the gambler runs out of money temporarily, but the gambler is allowed to continue gambling in the current loop iteration, and the program terminates only if he depletes his money when the program is back to the start of the outer loop. An invariant I for the program is as follows:

$$I(\ell) := \begin{cases} \textbf{true} & \text{if } \ell = 1 \\ x \geq 1 & \text{if } \ell = 2 \\ x \geq -8 \wedge 0 \leq y \leq 9 & \text{if } \ell = 3 \\ x \geq -7 \wedge 1 \leq y \leq 9 & \text{if } 4 \leq \ell \leq 11 \end{cases} .$$

For this program, we can define a DSM-map η as follows:

$$\eta(\ell, (x, y)) := \begin{cases} x & \text{if } \ell = 1 \\ x - 4/299 & \text{if } \ell = 2, 12 \\ x - 3/299 \cdot y + 7/299 & \text{if } \ell = 3 \\ x - 3/299 \cdot y + 3/299 & \text{if } \ell = 4 \\ x - 3/299 \cdot y - 1/299 & \text{if } \ell = 5, 8 \\ x - 3/299 \cdot y + 317/299 & \text{if } \ell = 6 \\ x - 3/299 \cdot y - 281/299 & \text{if } \ell = 7, 10 \\ x - 3/299 \cdot y + 616/299 & \text{if } \ell = 9 \\ x - 3/299 \cdot y + 14/299 & \text{if } \ell = 11 \end{cases} .$$

One can verify that η is a DSM-map by choosing $\epsilon = 4/299$, $a = -280/299$, $b = 617/299$ and $c = 1$. The minimal and maximal one-step differences of η are met in the transitions from labels 9 and 10 to label 11. Thus, the differences are in the interval $[-1 + 19/299, 2 + 19/299] = [a, b]$, and the expected value of η decreases by at least $4/299 = \epsilon$ in each step. Also, if the outer loop is not stopped, then $x \geq 1 = c$ at the initial label. The other conditions can be similarly checked. Thus, η is a DSM-map for P. Note that the internal while loop is a classic gambler's ruin and it is well-known that it terminates a.s. Therefore, by applying Theorem 6.4, we conclude that the program terminates a.s. under any initial valuation.

We now compare the notion of DSM-maps with RSMs/RSM-maps [Chakarov and Sankaranarayanan 2013; Chatterjee et al. 2016a,b] that have been successfully applied to prove finite expected termination time of probabilistic programs.

Remark 5 (Comparison with RSMs). *Our notion of DSM-maps is slightly (but crucially) different from RSMs [Chakarov and Sankaranarayanan 2013; Chatterjee et al. 2016a,b]. The difference is that a DSM-map does not require a global lower bound on its values, but instead requires the difference-boundedness condition, while an RSM requires its values to be non-negative, but has no difference-boundedness condition. As a result, the soundness of DSM-maps follows from concentration inequalities, while the soundness of RSMs follows from a limiting behavior of non-negative stochastic processes, a completely different aspect.*

Remark 6 (Comparison with [Fioriti and Hermanns 2015]). *We remark the reason why the approach in [Fioriti and Hermanns 2015] is not sound while ours is. This has to do with Crucial Issue 2 (Page 14). The approach in [Fioriti and Hermanns 2015] neglects the fact that RSMs have to be non-negative and is therefore not sound. In contrast, our approach uses DSM-maps which are not restricted to be non-negative and are sound for proving a.s. termination of probabilistic programs. As we have described previously, our approach of DSM-maps mainly strengthens the approach in [Fioriti and Hermanns 2015] with the strict decrease condition at every statement. The negativity of DSM-maps is then resolved through concentration inequalities, which guarantee that the probability of the value of R tending unboundedly to $-\infty$ is exponentially decreasing (see Lemma 6.2).*

In [Fioriti and Hermanns 2015], the authors describe their approach as compositional. *However, it requires compositional RSMs as a non-trivial side condition. In this work, we call such approaches* modular. *Note that our side conditions (DSMs) have the same modularity and complexity as the side conditions required by [Fioriti and Hermanns 2015] (compositional RSMs).*

Remark 7 (DSM-maps as a Debugging Tool). *Note that the DSM-maps contain a lot of useful information about the programs. For example, the verification could be "inverted" and used as a debugging tool, since DSM-maps can provide witnesses for proving/refuting a.s. termination. This point has also been mentioned in previous results on RepSMs (See [Chatterjee et al. 2017]). Given that our approach is modular and hence much faster for larger programs, it can also be used as an almost-real-time debugging tool and this point applies to it even more strongly than previous RepSM approaches.*

Remark 8 (Real-valued Variables). *Although we illustrate our approach on integer-valued variables, we show that it also works for real-valued variables. First, we directly extend the notion of DSM-maps to real-valued variables, where we only replace the discrete summation $\sum_{\mu \in Val_{V_r}} \overline{\Upsilon}(\mu) \cdot \eta(\ell', u(v, \mu))$ to an integral. Then we can prove the soundness of DSM-maps and construct the synthesis algorithm in the same way as for the integer case.*

7 A SOUND PROOF SYSTEM FOR ALMOST-SURE TERMINATION

In this section, we provide a proof system $\mathcal{D}$, in the style of Hoare logic, for the DSM approach. Note that DSM-maps treat the outer loop in a different manner than the inner loops. Specifically, in Definition 6.1, the requirement (D5) is only applied to the outer loop. This distinction is handled in our proof system by introducing different kinds of Hoare triples. Let R and R' be arithmetic expressions over program variables. The Hoare triple $\{R\}P\{R'\}$ indicates that for the program P, there exists a DSM-map η such that $\eta(\ell_{\text{in}}^P, \_) = R$ and $\eta(\ell_{\text{out}}^P, \_) = R'$. Similarly, the triple $\langle R\rangle P\langle R'\rangle$ indicates the existence of a map η with $\eta(\ell_{\text{in}}^P, \_) = R$ and $\eta(\ell_{\text{out}}^P, \_) = R'$ that satisfies all conditions of a DSM-map except for (D5). Intuitively, such an η is not a DSM-map for P, but it can be extended to a DSM-map for another program that contains P as an inner loop. We use the term *partial DSM* to describe such η.

We have following axiom schemata and rules in $\mathcal{D}$. Note that the DSM while rules are the most important novelty in our proof system. They are also the only pair of rules that are different for the two kinds of Hoare triples, i.e. the first while rule requires (D5), while the second while rule does not. Moreover, the values of a, b, c and ϵ are not fixed and can be different in every application of the rules below.

(1) DSM while rules:

$$\frac{\begin{array}{c}\langle R\rangle P\langle R'\rangle, \mathbf{G \longrightarrow R' \geq c},\\ G \longrightarrow a \leq R - R' \leq -\epsilon,\\ \text{and } \neg G \longrightarrow a \leq R'' - R' \leq -\epsilon\end{array}}{\{R'\}\ \textbf{while}\ G\ \textbf{do}\ P\ \textbf{od}\ \{R''\}}\ ,\qquad \frac{\begin{array}{c}\langle R\rangle P\langle R'\rangle,\\ G \longrightarrow a \leq R - R' \leq -\epsilon,\\ \text{and } \neg G \longrightarrow a \leq R'' - R' \leq -\epsilon\end{array}}{\langle R'\rangle\ \textbf{while}\ G\ \textbf{do}\ P\ \textbf{od}\ \langle R''\rangle}$$

(2) Skip statement axiom schemata:

$$\frac{a \leq R' - R \leq -\epsilon}{\{R\}\ \textbf{skip}\ \{R'\}}\ ,\qquad \frac{a \leq R' - R \leq -\epsilon}{\langle R\rangle\ \textbf{skip}\ \langle R'\rangle}$$

(3) Assignment axiom schemata:

$$\frac{a \leq \mathbb{E}[R'[x \leftarrow \mathfrak{e}]] - R \leq -\epsilon}{\{R\}\ x := \mathfrak{e}\ \{R'\}}\ ,\qquad \frac{a \leq \mathbb{E}[R'[x \leftarrow \mathfrak{e}]] - R \leq -\epsilon}{\langle R\rangle\ x := \mathfrak{e}\ \langle R'\rangle}$$

Here $R'[x \leftarrow \mathfrak{e}]$ is the expression obtained when one replaces all occurrences of the variable x in R' by the expression $\mathfrak{e}$.

(4) Sequential composition rules:

$$\frac{\{R\}P_1\{R'\}, \{R'\}P_2\{R''\}}{\{R\}P;Q\{R''\}} \quad , \quad \frac{\langle R\rangle P_1\langle R'\rangle, \langle R'\rangle P_2\langle R''\rangle}{\langle R\rangle P;Q\langle R''\rangle}$$

(5) Conditional branch rules:

$$\frac{\begin{array}{c}\{R_1\}P_1\{R'\}, \{R_2\}P_2\{R'\},\\ G \longrightarrow a \le R_1 - R \le -\epsilon,\\ \text{and } \neg G \longrightarrow a \le R_2 - R \le -\epsilon\end{array}}{\{R\}\ \textbf{if}\ G\ \textbf{then}\ P_1\ \textbf{else}\ P_2\{R'\}} \quad , \quad \frac{\begin{array}{c}\langle R_1\rangle P_1\langle R'\rangle, \langle R_2\rangle P_2\langle R'\rangle,\\ G \longrightarrow a \le R_1 - R \le -\epsilon,\\ \text{and } \neg G \longrightarrow a \le R_2 - R \le -\epsilon\end{array}}{\langle R\rangle\ \textbf{if}\ G\ \textbf{then}\ P_1\ \textbf{else}\ P_2\langle R'\rangle}$$

(6) Non-deterministic branch rules:

$$\frac{\begin{array}{c}\{R_1\}P_1\{R'\}, \{R_2\}P_2\{R'\},\\ a \le R_1 - R \le -\epsilon,\\ \text{and } a \le R_2 - R \le -\epsilon\end{array}}{\{R\}\ \textbf{if}\ \star\ \textbf{then}\ P_1\ \textbf{else}\ P_2\{R'\}} \quad , \quad \frac{\begin{array}{c}\langle R_1\rangle P_1\langle R'\rangle, \langle R_2\rangle P_2\langle R'\rangle,\\ a \le R_1 - R \le -\epsilon,\\ \text{and } a \le R_2 - R \le -\epsilon\end{array}}{\langle R\rangle\ \textbf{if}\ \star\ \textbf{then}\ P_1\ \textbf{else}\ P_2\langle R'\rangle}$$

(7) Probabilistic branch rules:

$$\frac{\begin{array}{c}\{R_1\}P_1\{R'\}, \{R_2\}P_2\{R'\},\\ a \le R_1 - R \le b,\\ a \le R_2 - R \le b,\\ \text{and } p \cdot R_1 + (1-p) \cdot R_2 \le R - \epsilon\end{array}}{\{R\}\ \textbf{if}\ \textbf{prob}(\mathrm{p})\ \textbf{then}\ P_1\ \textbf{else}\ P_2\{R'\}} \quad , \quad \frac{\begin{array}{c}\langle R_1\rangle P_1\langle R'\rangle, \langle R_2\rangle P_2\langle R'\rangle,\\ a \le R_1 - R \le b,\\ a \le R_2 - R \le b,\\ \text{and } p \cdot R_1 + (1-p) \cdot R_2 \le R - \epsilon\end{array}}{\langle R\rangle\ \textbf{if}\ \textbf{prob}(\mathrm{p})\ \textbf{then}\ P_1\ \textbf{else}\ P_2\langle R'\rangle}$$

The rules above can be used for establishing the existence of a DSM-map, which serves as a side condition in our modular approach for proving a.s. termination. Let Tm(P) denote that the program P terminates. The proof system $\mathcal{D}$ contains the following schemata and rules to modularly prove a.s. termination of programs:

(8) Modular DSM termination rule:

$$\frac{\begin{array}{c}\textsc{Tm}(P) \text{ and}\\ \{R_1\}\ \textbf{while}\ G\ \textbf{do}\ P\ \textbf{od}\ \{R_2\}\end{array}}{\textsc{Tm}(\ \textbf{while}\ G\ \textbf{do}\ P\ \textbf{od}\ )}$$

(9) Assignment and skip termination axiom schemata:

$$\frac{}{\textsc{Tm}(x := \mathfrak{e})} \quad , \quad \frac{}{\textsc{Tm}(\ \textbf{skip}\ )}$$

(10) Sequential composition termination rule:

$$\frac{\textsc{Tm}(P_1) \text{ and } \textsc{Tm}(P_2)}{\textsc{Tm}(P_1; P_2)}$$

(11) Branching composition termination rules:

$$\frac{\textsc{Tm}(P_1) \text{ and } \textsc{Tm}(P_2)}{\textsc{Tm}(\ \textbf{if}\ G\ \textbf{then}\ P_1\ \textbf{else}\ P_2\ \textbf{fi}\ )} \quad , \quad \frac{\textsc{Tm}(P_1) \text{ and } \textsc{Tm}(P_2)}{\textsc{Tm}(\ \textbf{if}\ \star\ \textbf{then}\ P_1\ \textbf{else}\ P_2\ \textbf{fi}\ )}$$

$$\frac{\textsc{Tm}(P_1) \text{ and } \textsc{Tm}(P_2)}{\textsc{Tm}(\ \textbf{if}\ \textbf{prob}(p)\ \textbf{then}\ P_1\ \textbf{else}\ P_2\ \textbf{fi}\ )}$$

Theorem 7.1. *The proof system $\mathcal{D}$ is sound for a.s. termination of probabilistic programs.*

Proof. Consider the first DSM while rule. Let Q = **while** G **do** P **od** . Suppose that $\langle R\rangle P\langle R'\rangle$ and the partial DSM of P is η, then we construct a DSM η' by defining $\eta'(\ell_{\text{in}}^{Q}, \_) := R'$, $\eta'(\ell_{\text{out}}^{Q}, \_) := R''$ and $\eta'(\ell, \_) := \eta(\ell, \_)$ for all labels ℓ in the loop body. It is easy to check that η' is a valid DSM, and we have $\{R'\}Q\{R''\}$. The soundness of the other DSM while rule can be proven in a similar manner. Rules (2)–(7) correspond to requirements (D1)–(D4) in the definition of a DSM-map (Definition 6.1). Note that it does not matter if different ϵ values were used to obtain the preconditions of these rules, given that one can use the smallest ϵ as the parameter for the DSM. The same point applies to a, b, c. Rule (8) is the same as Theorem 6.4. Rule (9) is sound because a single assignment or skip statement a.s. terminates. Soundness of rules (10)–(11) is proven in Lemma 3.2. □

We now argue from the perspective of proof rules that the approach of DSM-maps is a modular approach for proving a.s. termination. Note that in rule (8) above, we use the assumption that P terminates a.s., together with the side condition $\{R_1\}$ **while** G **do** P **od** $\{R_2\}$ (in the sense of Definition 2) to prove that **while** G **do** P **od** terminates a.s. as well. Also, it is worth mentioning that our approach does not synthesize a global RSM for the entire program. Instead, it finds distinct DSMs for each of the while loops. See [Huang et al. 2019, Appendix D] for a detailed example, in which two different DSM-maps are used for the internal and external while loops.

8 THE TEMPLATE-BASED ALGORITHM FOR SYNTHESIZING DSM-MAPS

In this section, we provide an efficient template-based algorithm for synthesizing linear DSM-maps. In theory, DSM-maps can take any general form. The only requirement is that they should satisfy (D1)-(D5) as in Definition 6.1. In this section, to synthesize a DSM-map from a given program, we first assume that the function has a special form, i.e. it is linear, and then establish constraints over its coefficients. Finally, the constraints can be solved through linear programming, leading to a sound method for generation of DSM-maps.

Recall that the existence of DSM-maps is used as a side condition in our modular approach for proving a.s. termination. Concretely, the algorithm provided in this section can replace rules (1)–(7) in the proof system $\mathcal{D}$. Hence, combining it with rules (8)–(11) leads to an efficient and completely automated modular method for proving a.s. termination.

Since DSM-maps are similar to RSM-maps [Chakarov and Sankaranarayanan 2013; Chatterjee et al. 2016a,b], we can directly extend previous algorithms for synthesizing linear/polynomial RSM-maps [Chakarov and Sankaranarayanan 2013; Chatterjee et al. 2016a,b] to linear DSM-maps. The key mathematical tool used in our algorithm is the well-known Farkas' Lemma.

Theorem (Farkas' Lemma [Farkas 1894; Schrijver 2003]). *Let* $\mathbf{A} \in \mathbb{R}^{m\times n}$, $\mathbf{b} \in \mathbb{R}^{m}$, $\mathbf{c} \in \mathbb{R}^{n}$ *and* $d \in \mathbb{R}$. *Assume that* $\{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A}\mathbf{x} \le \mathbf{b}\} \neq \emptyset$. *Then*

$$\{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A}\mathbf{x} \le \mathbf{b}\} \subseteq \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{c}^{\mathrm{T}}\mathbf{x} \le d\}$$

iff there exists $\mathbf{y} \in \mathbb{R}^m$ *such that* $\mathbf{y} \ge \mathbf{0}$, $\mathbf{A}^{\mathrm{T}}\mathbf{y} = \mathbf{c}$ *and* $\mathbf{b}^{\mathrm{T}}\mathbf{y} \le d$.

***The Farkas' Linear Assertions* Φ.** Farkas' Lemma transforms the inclusion testing of systems of linear inequalities into an emptiness problem. Given a polyhedron $H = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A}\mathbf{x} \le \mathbf{b}\}$ as in the statement of Farkas' Lemma (Theorem 8), we define the predicate $\Phi[H, \mathbf{c}, d](\xi)$ (which is called a Farkas' linear assertion) for Farkas' Lemma by

$$\Phi[H, \mathbf{c}, d](\xi) := (\xi \ge \mathbf{0}) \wedge \left(\mathbf{A}^{\mathrm{T}}\xi = \mathbf{c}\right) \wedge \left(\mathbf{b}^{\mathrm{T}}\xi \le d\right)$$

where ξ is a variable representing a column vector of dimension m.

Below, we fix an input probabilistic while loop P with a linear invariant I. We assume that P is *affine*, i.e. (i) every assignment statement in P has an affine expression at its right hand side; and

(ii) the loop guards of the conditional branches of P are in disjunctive normal form and each atomic proposition is a comparison between affine expressions.

The Synthesis Algorithm for DSM-maps. Our algorithm for synthesizing DSM-maps consists of the following four steps:

(1) *Template.* The algorithm establishes a template η for a DSM-map by setting $\eta(\ell, \mathbf{x}) := (\alpha_\ell)^{\mathrm{T}}\mathbf{x} + \beta_\ell$ for each $\ell \in L$ and $\mathbf{x} \in \mathbb{Z}^{|V_{\mathrm{p}}|}$, where α^ℓ is a vector of scalar variables and β^ℓ is a scalar variable, both representing unknown coefficients.

(2) *Variables for Parameters in a DSM-map.* The algorithm sets up a scalar variable ϵ, two scalar variables a, b and a scalar variable c. These variables directly correspond to the parameters for a DSM-map (see Definition 6.1).

(3) *Farkas' Linear Assertions.* From the template, we establish Farkas' linear assertions from the conditions (D1)–(D4). For example, the condition (D1) at a label ℓ requires that for the template η, it holds that $\sum_{\mu \in Val_{V_{\mathrm{r}}}} \overline{\Upsilon}(\mu) \cdot \eta(\ell', u(\nu, \mu)) \leq \eta(\ell, \nu) - \epsilon$ for all $\nu \in I(\ell)$. Since the template η is linear and we have affine assignments, the inequality $\sum_{\mu \in Val_{V_{\mathrm{r}}}} \overline{\Upsilon}(\mu) \cdot \eta(\ell', u(\nu, \mu)) \leq \eta(\ell, \nu) - \epsilon$ would also be linear. Then (D1) is essentially an inclusion of the set $I(\ell)$ in the halfspace represented by $\sum_{\mu \in Val_{V_{\mathrm{r}}}} \overline{\Upsilon}(\mu) \cdot \eta(\ell', u(\nu, \mu)) \leq \eta(\ell, \nu) - \epsilon$, and can be equivalently transformed into a group of Farkas' linear assertions, given that $I(\ell)$ is a finite union of polyhedra.

(4) *Solution through Linear Programming.* We group the constructed Farkas' linear assertions together in a conjunctive manner so that we have a system of linear inequalities over scalar variables (including template variables, parameter variables and fresh variables from Farkas' linear assertions). Then, we solve for the variables through linear programming. If we can get a solution for the scalar variables, then we get a DSM-map that witnesses the a.s. termination of the input program; otherwise, the algorithm cannot prove the a.s. termination property and outputs "*fail*".

Theorem 8.1. *Linear DSM-maps can be computed in polynomial time.*

Proof. It is straightforward to check that Steps (1)–(3) of the Synthesis algorithm have polynomial runtime. Hence, the resulting LP, which should be solved in Step (4), has polynomial size. It is well-known that LPs can be solved in polynomial time. □

Example 8.2. We now illustrate our synthesis algorithm on the program in Example 6.5.

- First, we set the template function $\eta(\ell, \mathbf{x}) = (\alpha_\ell)^{\mathrm{T}}\mathbf{x} + \beta_\ell$ for every label ℓ, where $\mathbf{x} = (x, y)^{\mathrm{T}}$ is the vector of program variables and the scalar variable β_ℓ together with the coordinate variables in the vector α_ℓ are unknown coefficients at a label ℓ.
- Second, we set up the parameters $\epsilon, a, b, c \in \mathbb{R}$ as in the definition of DSM-maps. The unknown coefficients in α^ℓ, β^ℓ and the parameters are what we want to solve for, in order to obtain a concrete DSM-map.
- In the third step, we establish Farkas' linear assertions. Below we illustrate an example on the construction of Farkas' linear assertions. Consider the condition (D4) at the label 5. The linear invariant at the label 5 is $x \geq -7 \wedge 1 \leq y \leq 9$ that represents the polyhedron $H = \left\{\mathbf{x} \mid \left(\begin{smallmatrix} -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{smallmatrix}\right)\mathbf{x} \leq \left(\begin{smallmatrix} 7 \\ 9 \\ -1 \end{smallmatrix}\right)\right\}$. To satisfy (D4), we have to ensure that the following conditions hold for every $\mathbf{x}$: $a \leq \eta(6, \mathbf{x}) - \eta(5, \mathbf{x}) \leq b$, $a \leq \eta(7, \mathbf{x}) - \eta(5, \mathbf{x}) \leq b$ and $\frac{6}{13}\eta(6, \mathbf{x}) + \frac{7}{13}\eta(7, \mathbf{x}) \leq \eta(5, \mathbf{x}) - \epsilon$. We first rewrite them into $\left(\alpha_5 - \alpha_6\right)^{\mathrm{T}}\mathbf{x} \leq -a + \beta_6 - \beta_5$, $(-\alpha_5 + \alpha_6)^{\mathrm{T}}\mathbf{x} \leq b - \beta_6 + \beta_5$ and $(\frac{6}{13}\alpha_6 + \frac{7}{13}\alpha_7 - \alpha_5)^{\mathrm{T}}\mathbf{x} \leq -\epsilon$. Let $d := -a + \beta_6 - \beta_5$, $d' := b - \beta_6 + \beta_5$. Then we construct the Farkas' linear assertions $\Phi[H, \alpha_5 - \alpha_6, d](\xi)$, $\Phi[H, -\alpha_5 + \alpha_6, d'](\xi')$ and $\Phi[H, \frac{6}{13}\alpha_6 + \frac{7}{13}\alpha_7 - \alpha_5, \epsilon](\xi'')$.
- Finally, in the fourth step we group all generated Farkas' linear assertions together in a conjunctive manner and solve for the unknown coefficients, together with the parameters

Program 1

```
1 : while $x \geq 1$ do
2 :     $z := y$;
3 :     while $z \geq 0$ do
4 :         $z := z - 1$;
5 :         $x := x + r$
        od;
6 :     $y := 2 \times y$;
7 :     $x := x - 1$
    od
8 :
```

Program 2

```
1 : while $x + y \geq 1$ do
2 :     $a := z$;
3 :     $b := z$;
4 :     while $a \geq 0$ do
5 :         $a := a - 1$;
6 :         $x := x + r_1$
        od;
7 :     while $b \geq 0$ do
8 :         $b := b - 1$;
9 :         $y := y + r_2$
        od;
10 :    $z := 2 \times z$;
11 :    $x := x - 1$
    od
12 :
```

Program 3

```
1 : while $x \geq 1$ do
2 :     $a := z$;
3 :     while $a \geq 0$ do
4 :         $a := a - 1$;
5 :         $b := z$;
6 :         while $b \geq 0$ do
7 :             $b := b - 1$;
8 :             $x := x + r$
            od;
9 :         $x := x + r$
        od;
10 :    $z := 2 \times z$;
11 :    $x := x - 1$
    od
12 :
```

Fig. 5. Our benchmark programs. These programs exhibit different types of nested while loops.

and the fresh variables from Farkas' linear assertions, using an LP-solver. If we can get a solution for the unknown coefficients, then the algorithm confirms that the input program is a.s. terminating (Theorem 6.4). Otherwise, the algorithm outputs "*fail*". In this case, our algorithm is able to synthesize a linear DSM-map (see Section 9).

9 EXPERIMENTAL RESULTS

In this section, we present experimental results obtained by an implementation of the algorithm of Section 8. Note that our algorithm has very few dependencies, all of which are standard operations (e.g. linear invariant generation and linear programming).

Experimental Benchmarks. We consider two families of benchmarks:

- First, to illustrate the applicability of our approach to different types of while loops, we consider the program of Figure 2 (i.e. the counterexample to the FHV-rule), the Mini-roulette program of Example 6.5, and three other classical examples of probabilistic programs that exhibit various types of nested while loops (Figure 5). *Program 1* is a simple nested while loop, in which the outer loop control variable is updated in the inner loop. *Program 2* is a nested while loop with two sequentially-composed inner loops, in which the outer loop control variables are each updated in one of these inner loops. *Program 3* is a three-level nested while loop.
- Second, we demonstrate that our approach can handle real-world programs by providing experimental results on the benchmarks used in [Ngo et al. 2018].

Invariants. Our approach is able to synthesize DSMs using very simple invariants obtained from the loop guards. See [Huang et al. 2019, Appendix E.1] for more details. Note that in all cases, the invariants we use are strictly weaker than, and can be replaced by, invariants generated by standard tools such as [Colón et al. 2003] and [Sankaranarayanan et al. 2004]. However, we use weaker invariants to demonstrate the power of our algorithm.

Distributions. We assume that each sampling variable r in Programs 1, 2 and 3 is sampled according to the distribution $\mathbb{P}(r = 1) = 0.25, \mathbb{P}(r = -1) = 0.75$. This choice is arbitrary and our approach can

Table 1. Experimental Results on Example Programs

Example	Result	Runtime (s)	$\eta(\ell_{\text{in}})$	$[a, b]$
Counterexample	Failure	0.774	–	–
Example 6.5	Success	0.812	$75.4 \cdot x$	$[-70.6, 155.6]$
Program 1	Success	0.722	$6 \cdot x + 5$	$[-4, 8]$
Program 2	Success	0.921	$7 \cdot x + 7 \cdot y + 6$	$[-5, 9.5]$
Program 3	Success	0.872	$8 \cdot x + 7$	$[-5, 11]$

synthesize linear DSMs for any distribution, as long as such a DSM exists. The benchmarks of [Ngo et al. 2018] contain a specification of the distributions.

Implementation and Experiment Machine. We implemented our approach in Java. Our implementation successfully passed the OOPSLA artifact evaluation and is available at http://pub.ist.ac.at/~akafshda/DSM/. We used lpsolve [Berkelaar et al. 2004] and JavaILP [Lukasiewycz 2008] for solving the linear programming instances. The results were obtained on a Windows 10 machine with a 2.5 GHz Intel Core i5-2520M processor and 8 GB of RAM.

Experimental Results. Table 1 summarizes our experimental results over the five example programs and Table 2 provides the results over the benchmarks from [Ngo et al. 2018]. Note that the counterexample program does not terminate almost surely. Therefore, any sound approach is expected to fail on this program. In all other cases, our approach is extremely efficient. It processes each benchmark program in less than 2 seconds and successfully synthesizes *separate* linear DSM-maps for each of the while loops in the program. In all cases, the DSM parameters ϵ and c are synthesized as 1 and 0, respectively (except that $c = -71$ for coupon). The reported runtime for each benchmark is the total time spent for synthesizing DSM-maps for all while loops in the benchmark. The column $\eta(\ell_{\text{in}})$ reports the expression at the first label ℓ_{in} of the program in the DSM-map η corresponding to the outermost loop. See [Huang et al. 2019, Appendix E.2] for more details.

10 RELATED WORKS

We compare our results with the most related works on termination verification of probabilistic programs. We discuss two main classes of approaches: supermartingale-based and proof-rule-based.

Supermartingale-based approaches. The most related supermartingale-based works are [Bournez and Garnier 2005; Chakarov and Sankaranarayanan 2013; Chatterjee and Fu 2019; Chatterjee et al. 2016a,b, 2017; Fioriti and Hermanns 2015; McIver and Morgan 2004, 2005; McIver et al. 2018]. Compared to these results, the most significant difference is that our result considers modular verification of the termination property, while previous approaches tackle the termination problem directly on the whole program (except the cases mentioned below). In detail, we synthesize individual DSMs for each loop in the program, while most previous results synthesize a global (ranking) supermartingale for the whole program and do not have the modular feature.

Another advantage of our approach is that we do not require non-negativity of supermartingales, which is however required in all of the previous results mentioned above. For example, consider Program 1 in Figure 5. In this example, we have a DSM-map for the outer loop that only involves the program variable x (see Table 1 and [Huang et al. 2019, Appendix E.2]). Observe that (i) the expected value of x decreases throughout the outer loop, and (ii) the value of x is unbounded and can become arbitrarily positive or negative. Also note that the decrease in x is the main reason that the loop terminates a.s. In previous approaches, due to the restrictive requirement of non-negativity,

Table 2. Experimental Results on the Benchmarks of [Ngo et al. 2018]. Benchmarks that contain nested loops are marked with a ◊.

Benchmark	Result	Runtime (s)	$\eta(\ell_{\text{in}})$	$[a, b]$
`ber`	Success	0.597	$4 \cdot n - 4 \cdot x + 1$	$[-3, 1]$
`bin`	Success	0.822	$0.4 \cdot n - 0.4 \cdot x + 1$	$[-3, 1]$
`C4B_t09`	Success	0.665	$-4 \cdot j + 4 \cdot x + 1$	$[-1, 0]$
`C4B_t13` ◊	Success	1.308	$4.5 \cdot x + 2 \cdot y - 1$	$[-1.5, 0.5]$
`C4B_t15` ◊	Success	1.264	$5 \cdot x + 2$	$[-5, 0]$
`C4B_t19`	Success	1.202	$6 \cdot i + 5$	$[-4, 2]$
`C4B_t30`	Success	0.653	$2.5 \cdot x + 2.5 \cdot y + 1$	$[-3.6, 1.5]$
`C4B_t61`	Success	1.217	$0.286 \cdot l$	$[-1.3, 0]$
`complex` ◊	Success	1.444	$8 \cdot N - 8 \cdot x + 1$	$[-5, 3]$
`condand`	Success	0.637	$3 \cdot m + 3 \cdot n + 1$	$[-1, 0]$
`cooling` ◊	Success	1.364	$2 \cdot mt - 2 \cdot st + 2.443 \cdot pt + 3.821$	$[-2.6, 0]$
`coupon`	Success	0.769	$-35 \cdot i + 69$	$[-29, 27]$ $c = -71$
`cowboy_duel`	Success	0.632	$4.066 \cdot \text{flag} - 1.162$	$[-2.7, 2.2]$
`filling_vol`	Success	0.720	$-3.356 \cdot \text{volMeasured} + 3.356 \cdot \text{volToFill} + 3$	$[-16.6, 2.3]$
`geo`	Success	0.628	$7 \cdot \text{flag} + 1$	$[-4, 2]$
`hyper`	Success	0.599	$10 \cdot n - 10 \cdot x$	$[-19, 1]$
`linear01`	Success	0.614	$1.796 \cdot x + 2.593$	$[-1.6, 0.2]$
`prdwalk`	Success	0.661	$1.770 \cdot n - 1.770 \cdot x + 1$	$[-5.6, 3.2]$
`prnes` ◊	Success	1.328	$-23.655 \cdot n + 0.032 \cdot y + 4.365$	$[-17.3, 20.3]$
`prseq`	Success	1.242	$1.005 \cdot x - 1.005 \cdot y + 1$	$[-2, 0]$
`prspeed`	Success	0.947	$8 \cdot m + 4.161 \cdot n - 4.161 \cdot x - 8 \cdot y + 7.484$	$[-5, 3]$
`race`	Success	0.687	$-3.279 \cdot h + 3.279 \cdot t + 14.557$	$[-17.2, 15.6]$
`rdseql` ◊	Success	1.261	$5.5 \cdot x + 2 \cdot y - 2$	$[-1.5, 0.5]$
`rdspeed`	Success	0.760	$6 \cdot m + 2 \cdot n - 2 \cdot x - 6 \cdot y + 2$	$[-4, 2]$
`rdwalk`	Success	0.625	$6 \cdot n - 6 \cdot x + 1$	$[-10, 8]$
`rfind_lv`	Success	0.625	$6 \cdot \text{flag} + 1$	$[-4, 2]$
`rfind_mc`	Success	0.668	$0.5 \cdot \text{flag} - 3.75 \cdot i + 3.75 \cdot k + 1.25$	$[-1.25, 0]$
`robot`	Success	1.562	$28.778 \cdot N + 114.444$	$[-216.7, 1.7]$
`roulette`	Success	0.905	$27.595 \cdot \text{money} + 205.962$	$[-205, 109.4]$
`sprdwalk`	Success	0.693	$4 \cdot n - 4 \cdot x + 1$	$[-3, 1]$
`trapped_miner` ◊	Success	1.676	$-9 \cdot i + 9 \cdot n + 8$	$[-6, 4]$

we cannot choose x as a (ranking) supermartingale. Moreover, we cannot choose $|x|$ either, given that the expected value of $|x|$ increases at $x = 0$. Hence, it is non-trivial to obtain a *non-negative* (ranking) supermartingale for this example. In contrast, our approach is more flexible and succinctly proves the a.s. termination property of this program by synthesizing two distinct DSM-maps: a DSM on z for the inner loop and a DSM on x for the outer loop.

We now compare our approach to previous supermartingale-based results that provide some level of modularity.

Comparison with [Fioriti and Hermanns 2015]. This is the most similar result. However, we have already shown that this approach is not sound. We have also presented the minimal required strengthening and proven that our new approach is sound.

Comparison with [McIver et al. 2018]. Although the approach in [McIver et al. 2018] is also modular and constructs supermartingales loop-by-loop, their approach has the following disadvantages: (i) their approach is restricted to non-negative supermartingales, and cannot be used when a non-negative supermartingale is hard to construct (see our above point about Program 1 in

Figure 5); (ii) their approach requires to calculate the complete semantics of the loop body, which is infeasible in general, while our approach (together with our algorithmic method) only requires to examine the syntax of the loop body.

Comparison with [Agrawal et al. 2018]. Another related result in [Agrawal et al. 2018] considers lexicographic RSMs that are sound for a.s. termination of probabilistic programs. While lexicographic RSMs have some flavor of modularity (such as decomposition based on lexicographic order), they also synthesize a global lexicographic RSM and hence are not modular in the sense of (2). Moreover, their approach also requires the non-negativity of lexicographic RSMs, thus suffers the same problem when constructing non-negative RSMs is difficult.

Proof-rule-based approaches. Another family of approaches for termination analysis are based on the notion of proof rules [Hesselink 1993; Jones 1989; Kaminski et al. 2016; McIver et al. 2018; Olmedo et al. 2016]. For example, [Kaminski et al. 2016] presents a proof-rule-based approach for proving finite expected termination time of probabilistic while loops, and [Olmedo et al. 2016] presents sound proof rules for probabilistic programs with recursion. Another related work is [Batz et al. 2019], which focuses on memory safety, while our focus is on termination. Most results on proof rules focus on specifying local logical properties at every label to ensure a global logical property, and do not consider modular proof rules. In contrast, we provide modular proof rules that prove the almost-sure termination property. The most relevant result is given in [Ngo et al. 2018] that presents a modular approach for deriving resource bounds of probabilistic programs. Compared with our result, their result focuses on resource bounds and can only handle programs with finite expected resource consumption, whereas our result focuses on termination properties and can handle programs with infinite expected termination time. An example can be obtained by considering Program 1, Figure 5 and changing $z := z - 1$ at label 4 to $z := z + r'$, where we have $\mathbb{P}(r' = 1) = \mathbb{P}(r' = -1) = 0.5$. Then the inner loop models a symmetric walk that terminates a.s. but with infinite expected termination time. Therefore, this program has infinite expected termination time. For this modified example, the original DSM-map remains valid (see "Program 1" in Table 1 and [Huang et al. 2019, Appendix E.2]). Thus, our modular approach proves its a.s. termination. Our approach only relies on a side condition (existence of a DSM) and the assumption that the loop body is a.s. terminating, so it can handle loop bodies with infinite expected termination time.

11 CONCLUSION

In this paper, we first proved that a natural probabilistic extension of the variant rule in the Floyd-Hoare logic is not sound for modular verification of almost-sure termination of probabilistic programs and identified the flaw in the previous related work [Fioriti and Hermanns 2015]. Then, we proposed a minimal sound strengthening of the approach in [Fioriti and Hermanns 2015] through the notion of descent supermartingales (DSMs), and demonstrated an efficient algorithmic implementation of our strengthened approach for linear DSMs. An important future direction is to investigate different rules and sound approaches for modular verification of probabilistic termination. Another direction is to consider the algorithmic problem of synthesizing non-linear DSM-maps.

ACKNOWLEDGMENTS

The research was partially supported by the National Natural Science Foundation of China (Grant No. 61772336, 61802254), Open Project of Shanghai Key Laboratory of Trustworthy Computing, Vienna Science and Technology Fund (WWTF) Project ICT15-003, Austrian Science Fund (FWF) NFN Grant No. S11407-N23 (RiSE/SHiNE), ERC Starting Grant (279307: Graph Games), an IBM PhD Fellowship, and DOC Fellowship No. 24956 of the Austrian Academy of Sciences (ÖAW). We are sincerely grateful to Prof. Yuxi Fu, director of the support of the BASICS Lab at Shanghai Jiao Tong University, for his support.

REFERENCES

Sheshansh Agrawal, Krishnendu Chatterjee, and Petr Novotný. 2018. Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs. *PACMPL* 2, POPL (2018), 34:1–34:32.

Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. MIT Press.

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. 2019. Quantitative separation logic: a logic for reasoning about probabilistic pointer programs. In *POPL*.

Michel Berkelaar, Kjell Eikland, Peter Notebaert, et al. 2004. lpsolve: Open source (mixed-integer) linear programming system. *Eindhoven U. of Technology* (2004).

Olivier Bournez and Florent Garnier. 2005. Proving Positive Almost-Sure Termination. In *RTA*. 323–337.

Aleksandar Chakarov and Sriram Sankaranarayanan. 2013. Probabilistic Program Analysis with Martingales. In *CAV*. 511–526.

Krishnendu Chatterjee and Hongfei Fu. 2019. Termination of Nondeterministic Recursive Probabilistic Programs. In *VMCAI*.

Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2016a. Termination Analysis of Probabilistic Programs Through Positivstellensatz's. In *CAV*. 3–22.

Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Nastaran Okati. 2018. Computational Approaches for Stochastic Shortest Path on Succinct MDPs. In *IJCAI 2018*. 4700–4707.

Krishnendu Chatterjee, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. 2016b. Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In *POPL*. 327–342.

Krishnendu Chatterjee, Petr Novotný, and Ðorđe Žikelić. 2017. Stochastic invariants for probabilistic termination. In *POPL*. 145–160.

Guillaume Claret, Sriram K Rajamani, Aditya V Nori, Andrew D Gordon, and Johannes Borgström. 2013. Bayesian inference using data flow analysis. In *Joint Meeting on Foundations of Software Engineering*. ACM, 92–102.

Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. 2003. Linear Invariant Generation Using Non-linear Constraint Solving. In *CAV*. 420–432.

Javier Esparza, Andreas Gaiser, and Stefan Kiefer. 2012. Proving Termination of Probabilistic Programs Using Patterns. In *CAV*. 123–138.

Kousha Etessami and Mihalis Yannakakis. 2009. Recursive Markov chains, stochastic grammars, and monotone systems of nonlinear equations. *Journal of the ACM (JACM)* 56, 1 (2009), 1.

Julius Farkas. 1894. A Fourier-féle mechanikai elv alkalmazásai (Hungarian). *Mathematikaiés Természettudományi Értesitö* 12 (1894), 457–472.

Luis María Ferrer Fioriti and Holger Hermanns. 2015. Probabilistic Termination: Soundness, Completeness, and Compositionality. In *POPL*. 489–501.

Robert W. Floyd. 1967. Assigning meanings to programs. *Mathematical Aspects of Computer Science* 19 (1967), 19–33.

Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. 2016. Probabilistic NetKAT. In *ESOP*. Springer, 282–309.

Noah D Goodman, Vikash K Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B Tenenbaum. 2008. Church: a language for generative models. In *UAI*. AUAI Press, 220–229.

Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. (2014).

Andrew D Gordon, Mihhail Aizatulin, Johannes Borgstrom, Guillaume Claret, Thore Graepel, Aditya V Nori, Sriram K Rajamani, and Claudio Russo. 2013. A model-learner pattern for Bayesian reasoning. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 403–416.

Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. 2014. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*. ACM, 167–181.

Wim H. Hesselink. 1993. Proof Rules for Recursive Procedures. *Formal Asp. Comput.* 5, 6 (1993), 554–570.

Wassily Hoeffding. 1963. Probability Inequalities for Sums of Bounded Random Variables. *J. Amer. Statist. Assoc.* 58, 301 (1963), 13–30.

Mingzhang Huang, Hongfei Fu, and Krishnendu Chatterjee. 2018. New Approaches for Almost-Sure Termination of Probabilistic Programs. In *APLAS*. 181–201.

Mingzhang Huang, Hongfei Fu, Krishnendu Chatterjee, and Amir Kafshdar Goharshady. 2019. Modular Verification for Almost-Sure Termination of Probabilistic Programs. *arXiv preprint arXiv:1901.06087* (2019).

Claire Jones. 1989. *Probabilistic Non-Determinism*. Ph.D. Dissertation. The University of Edinburgh.

David M. Kahn. 2017. Undecidable Problems for Probabilistic Network Programming. In *MFCS*. 68:1–68:17.

Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2016. Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs. In *ESOP*. 364–389.

Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2018. On the hardness of analyzing probabilistic programs. *Acta Informatica* (2018), 1–31.

Shmuel Katz and Zohar Manna. 1975. A Closer Look at Termination. *Acta Inf.* 5 (1975), 333–352. https://doi.org/10.1007/BF00264565

Orna Kupferman and Moshe Y. Vardi. 1997. Modular Model Checking. In *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany, September 8-12, 1997. Revised Lectures (Lecture Notes in Computer Science)*, Willem P. de Roever, Hans Langmaack, and Amir Pnueli (Eds.), Vol. 1536. Springer, 381–401. https://doi.org/10.1007/3-540-49213-5_14

Martin Lukasiewycz. 2008. JavaILP - Java Interface to ILP Solvers, http://javailp.sourceforge.net/. (2008). http://javailp.sourceforge.net/

Hosam Mahmoud. 2008. *Pólya urn models.* Chapman and Hall/CRC.

Christopher D Manning, Christopher D Manning, and Hinrich Schütze. 1999. *Foundations of statistical natural language processing.* MIT press.

Colin McDiarmid. 1998. Concentration. In *Probabilistic Methods for Algorithmic Discrete Mathematics.* 195–248.

Annabelle McIver and Carroll Morgan. 2004. Developing and Reasoning About Probabilistic Programs in *pGCL*. In *PSSE.* 123–155.

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

Annabelle McIver, Carroll Morgan, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. 2018. A new proof rule for almost-sure termination. *PACMPL* 2, POPL (2018), 33:1–33:28. https://doi.org/10.1145/3158121

Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded expectations: resource analysis for probabilistic programs. In *PLDI.* 496–512.

Federico Olmedo, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2016. Reasoning about Recursive Probabilistic Programs. In *LICS.* 672–681.

DM Roy, VK Mansinghka, ND Goodman, and JB Tenenbaum. 2008. A stochastic programming perspective on nonparametric Bayes. In *Nonparametric Bayesian Workshop, Int. Conf. on Machine Learning*, Vol. 22. 26.

Sriram Sankaranarayanan, Henny B Sipma, and Zohar Manna. 2004. Constraint-based linear-relations analysis. In *SAS 2004.* Springer, 53–68.

Alexander Schrijver. 2003. *Combinatorial Optimization - Polyhedra and Efficiency.* Springer.

Adam Ścibior, Zoubin Ghahramani, and Andrew D Gordon. 2015. Practical probabilistic programming with monads. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 165–176.

Steffen Smolka, Praveen Kumar, Nate Foster, Dexter Kozen, and Alexandra Silva. 2017. Cantor meets Scott: semantic foundations for probabilistic networks. In *POPL.* 557–571.

Sebastian Thrun. 2000. Probabilistic algorithms in robotics. *Ai Magazine* 21, 4 (2000), 93.

Sebastian Thrun. 2002. Probabilistic robotics. *Commun. ACM* 45, 3 (2002), 52–57.

David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. 2016. Design and Implementation of Probabilistic Programming Language Anglican. In *IFL.* ACM, 6:1–6:12. https://doi.org/10.1145/3064899.3064910

Di Wang, Jan Hoffmann, and Thomas W. Reps. 2018. PMAF: an algebraic framework for static analysis of probabilistic programs. In *PLDI.* 513–528.

Peixin Wang, Hongfei Fu, Amir Kafshdar Goharshady, Krishnendu Chatterjee, Xudong Qin, and Wenjun Shi. 2019. Cost Analysis of Nondeterministic Probabilistic Programs. In *PLDI.* 204–220.

David Williams. 1991. *Probability with Martingales.* Cambridge University Press.