HAL
open science

Synthesizing Probabilistic Invariants via Doob's Decomposition

Gilles Barthe, Thomas Espitau, Luis María Ferrer Fioriti, Justin Hsu

To cite this version:

Gilles Barthe, Thomas Espitau, Luis María Ferrer Fioriti, Justin Hsu. Synthesizing Probabilistic Invariants via Doob's Decomposition. 28th International Conference on Computer Aided Verification , Jul 2016, Toronto, Canada. pp.43-61, 10.1007/978-3-319-41528-4_3 . hal-01561429

HAL Id: hal-01561429

https://hal.science/hal-01561429

Submitted on 12 Jul 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Synthesizing Probabilistic Invariants via Doob's Decomposition

Gilles Barthe[1], Thomas Espitau[2], Luis María Ferrer Fioriti[3], and Justin Hsu[4]

[1] IMDEA Software Institute [2] ENS Cachan
[3] Saarland University [4] University of Pennsylvania

Abstract. When analyzing probabilistic computations, a powerful approach is to first find a *martingale*—an expression on the program variables whose expectation remains invariant—and then apply the optional stopping theorem in order to infer properties at termination time. One of the main challenges, then, is to systematically find martingales.
We propose a novel procedure to synthesize martingale expressions from an arbitrary initial expression. Contrary to state-of-the-art approaches, we do not rely on constraint solving. Instead, we use a symbolic construction based on *Doob's decomposition*. This procedure can produce very complex martingales, expressed in terms of conditional expectations.
We show how to *automatically* generate and simplify these martingales, as well as how to apply the *optional stopping theorem* to infer properties at termination time. This last step typically involves some simplification steps, and is usually done manually in current approaches. We implement our techniques in a prototype tool and demonstrate our process on several classical examples. Some of them go beyond the capability of current semi-automatic approaches.

1 Introduction

Probabilistic computations are a key tool in modern computer science. They are ubiquitous in machine learning, privacy-preserving data mining, cryptography, and many other fields. They are also a common and flexible tool to model a broad range of complex real-world systems. Not surprisingly, probabilistic computations have been extensively studied from a formal verification perspective. However, their verification is particularly challenging.

In order to understand the difficulty, consider the standard way to infer properties about the final state of a non-probabilistic program using a strong invariant (an assertion which is preserved throughout program execution) and a proof of termination. This proof principle is not easily adapted to the probabilistic case. First, probabilistic programs are interpreted as distribution transformers [25] rather than state transformers. Accordingly, assertions (including strong invariants) must be interpreted over distributions. Second, the notion of termination is different for probabilistic programs. We are usually not interested in proving that *all* executions are finite, but merely that the probability of termination is 1, a slightly weaker notion. Under this notion, there may be no finite bound on

the number of steps over all possible executions. So, we cannot use induction to transfer local properties to the end of the program—more complex limiting arguments are needed.

We can avoid some of these obstacles by looking at the *average* behavior of a program. That is, we can analyze numerical expressions (over program variables) whose average value is preserved. These expressions are known as martingales, and have several technical advantages. First, martingales are easy to manipulate symbolically and can be checked locally. Second, the average value of martingale is preserved at termination, even if the control-flow of the program is probabilistic. This fact follows from the *optional stopping theorem* (OST), a powerful result in martingale theory.

While martingales are quite useful, they can be quite non-obvious. Accordingly, recent investigation has turned to automatically synthesizing martingales. State-of-the-art frameworks are based on constraint solving, and require the user to provide either a template expression [22, 6] or a limit on the search space [10, 7]. The main advantage of such approaches is that they are generally complete—they find *all* possible martingales in the search space. However, they have their drawbacks: a slightly wrong template can produce no invariant at all, and a lot of search space may be needed to arrive at the martingale.

We propose a framework that *complements* current approaches—we rely on purely symbolic methods instead of solving constraints or searching. We require the user to provide a "seed" expression, from which we *always* generate a martingale. Our approach uses *Doob's decomposition theorem*, which gives a symbolic method to construct a martingale from any sequence of random values. Once we have the martingale, we can apply optional stopping to reason about the martingale at loop termination. While the martingale and final fact may be quite complex, we can use non-probabilistic invariants and symbolic manipulations to automatically simplify them.

We demonstrate our techniques in a prototype implementation, implementing Doob's decomposition and the Optional Stopping Theorem. Although these proof principles have been long known to probability theory, we are the first to incorporate them into an automated program analysis. Given basic invariants and hints, our prototype generates martingales and facts for a selection of examples.

2 Mathematical preliminaries

We briefly introduce some definitions from probability theory required for our technical development. We lack the space to discuss the definitions in-depth, but we will explain informally what the various concepts mean in our setting. Interested readers can find a more detailed presentation in any standard probability theory textbook (e.g., Williams [33]).

First, we will need some basic concepts from probability theory.

Definition 1 *Let Ω be the set of outcomes.*

- *A* sigma algebra *is a set $\mathcal{F}$ of subsets of Ω, closed under complements and countable unions, and countable intersections.*

– *A* probability measure *is a countably additive mapping* $\mathbb{P} : \mathcal{F} \to [0, 1]$ *such that* $\mathbb{P}(\Omega) = 1$.
– *A* probability *space is a triple* $(\Omega, \mathcal{F}, \mathbb{P})$.

Next, we can formally define stochastic processes. These constructions are technical but completely standard.

Definition 2 *Let* $(\Omega, \mathcal{F}, \mathbb{P})$ *be a probability space.*

– *A* (real) random variable *is a function* $X : \Omega \to \mathbb{R}$. X *is* $\mathcal{F}$-measurable *if* $X^{-1}((a, b]) \in \mathcal{F}$ *for every* $a, b \in \mathbb{R}$.
– *A* filtration *is a sequence* $\{\mathcal{F}_i\}_{i\in\mathbb{N}}$ *of sigma algebras such that* $\mathcal{F}_i \subseteq \mathcal{F}$ *and* $\mathcal{F}_{i-1} \subseteq \mathcal{F}_i$ *for every* $i > 0$. *When there is a fixed filtration on* $\mathcal{F}$, *we will often abuse notation and write* $\mathcal{F}$ *for the filtration.*
– *A* stochastic process *adapted to filtration* $\mathcal{F}$ *is a sequence of random variables* $\{X_i\}_{i\in\mathbb{N}}$ *such that each* X_i *is* $\mathcal{F}_i$*-measurable.*

Intuitively, we can think of Ω as a set where each element represents a possible outcome of the samples. In our setting, grouping samples according to the loop iteration gives a natural choice for the filtration: we can take $\mathcal{F}_i$ to be the set of events that are defined by samples in iteration i or before. A stochastic process X is adapted to this filtration if X_i is defined in terms of samples from iteration i or before. Sampled variables at step i are independent of previous steps, so they are not $\mathcal{F}_{i-1}$-measurable.

Expectation. To define martingales, we need to introduce expected values and conditional expectations. The *expected value* of a random variable is defined as

$$\mathbf{E}[X] \triangleq \int_{\Omega} X \cdot \mathrm{d}\mathbb{P}$$

where $\int$ is the Lebesgue integral [33]. We say that a random variable is *integrable* if $\mathbf{E}[|X|]$ is finite. Given a integrable random variable X and a sigma algebra $\mathcal{G}$, a *conditional expectation* of X with respect to $\mathcal{G}$ is a random variable Y such that Y is $\mathcal{G}$-measurable, and $\mathbf{E}[X \cdot \mathbb{1}_{\{A\}}] = \mathbf{E}[Y \cdot \mathbb{1}_{\{A\}}]$ for all events $A \in \mathcal{G}$. (Recall that the *indicator function* $\mathbb{1}_{\{A\}}$ of an event A maps $\omega \in A$ to 1, and all other elements to 0.) Since one can show that this Y is essentially unique, we denote it by $\mathbf{E}[X \mid \mathcal{G}]$.

Moments. Our method relies on computing higher-order moments. Suppose X is a random variable with distribution d. If X takes numeric values, the *kth moment* of d is defined as

$$G(d)_k \triangleq \mathbf{E}[X^k]$$

for $k \in \mathbb{N}$. If X ranges over tuples, the *correlations* of d are defined as

$$G(d, \{a, b\})_{p,q} \triangleq \mathbf{E}[\pi_a(X)^p \cdot \pi_b(X)^q],$$

for $p, q \in \mathbb{N}$, and similarly for products of three or more projections. Here, the *projection* $\pi_i(X)$ for X a tuple-valued random variable is the marginal distribution of the ith coordinate of the tuple.

Martingales. A martingale is a stochastic process with a special property: the average value of the current step is equal to the value of the previous step.

Definition 3 *Let $\{X_i\}$ be a stochastic process adapted to filtration $\{\mathcal{F}_i\}$. We say that X is a* martingale *with respect to $\mathcal{F}$ if it satisfies the property*

$$\mathbf{E}[X_i \mid \mathcal{F}_{i-1}] = X_{i-1}.$$

For a simple example, consider a symmetric random walk on the integers. Let $X \in \mathbb{Z}$ denote the current position of the walk. At each step, we flip a fair coin: if heads, we increase the position by 1, otherwise we decrease the position by 1. The sequence of positions $X_0, X_1, \ldots$ forms a martingale since the average position at time i is simply the position at time $i-1$:

$$\mathbf{E}[X_i \mid \mathcal{F}_{i-1}] = X_{i-1}.$$

Doob's decomposition. One important result in martingale theory is Doob's decomposition. Informally, it establishes that any integrable random process can be written uniquely as a sum of a martingale and a predictable process. For our purposes, it gives a constructive and purely symbolic method to extract a martingale from any arbitrary random process.

Theorem 1 (Doob's decomposition) *Let $X = \{X_i\}_{i\in\mathbb{N}}$ be a stochastic process adapted to filtration $\{\mathcal{F}_i\}_{i\in\mathbb{N}}$ where each X_i has finite expected value. Then, the following process is a martingale:*

$$M_i = \begin{cases} X_0 & : i = 0 \\ X_0 + \sum_{j=1}^{i} X_j - \mathbf{E}[X_j \mid \mathcal{F}_{j-1}] & : i > 0 \end{cases}$$

If X is already a martingale, then $M = X$.

We will think of the stochastic process X as a seed process which generates the martingale. While the definition of the martingale involves conditional expectations, we will soon see how to automatically simplify these expectations.

Optional stopping theorem. For any martingale M, it is not hard to show that the expected value of M remains invariant at each time step. That is, for any fixed value $n \in \mathbb{N}$, we have

$$\mathbf{E}[M_n] = \mathbf{E}[M_0].$$

The optional stopping theorem extends this equality to situations where n itself may be random, possibly even a function of the martingale.

Definition 4 *Let $(\Omega, \mathcal{F}, \mathbb{P})$ be a probability space with filtration $\{\mathcal{F}_i\}_{i\in\mathbb{N}}$. A random variable $\tau : \Omega \to \mathbb{N}$ is a* stopping time *if the subset $\{w \in \Omega \mid \tau(w) \leq i\}$ is a member of $\mathcal{F}_i$ for each $i \in \mathbb{N}$.*

Returning to our random walk example, the first time that the position is farther than 100 from the origin is a stopping time since this time depends only on past samples. In contrast, the last time that a position is farther than 100 from the origin is *not* a stopping time, since this time depends on future samples. More generally, the iteration count when we exit a probabilistic loop is a stopping time since termination is a function of past samples only.

If we have a stopping time and a few mild conditions, we can apply the optional stopping theorem.[1]

Theorem 2 (Optional stopping) *Let τ be a stopping time, and let M be a martingale. If the expected value of τ is finite, and if $|M_i - M_{i-1}| \leq C$ for all $i > 0$ and some constant C, then*

$$\mathbf{E}[M_\tau] = \mathbf{E}[M_0].$$

To see this theorem in action, consider the random walk martingale S and take the stopping time τ to be the first time that $|S| \geq 100$. It is possible to show that τ has finite expected value, and clearly $|S_i - S_{i-1}| \leq 1$. So, the optional stopping theorem gives

$$\mathbf{E}[S_\tau] = \mathbf{E}[S_0] = 0.$$

Since we know that the position is ± 100 at time τ, this immediately shows that the probability of hitting $+100$ is equal to the probability of hitting -100. This intuitive fact can be awkward to prove using standard probabilistic invariants, but falls out directly from a martingale analysis.

3 Overview of method

Now that we have seen the key technical ingredients of our approach, let us see how to combine these tools into an automated program analysis. We will take an imperative program specifying a stochastic process and a seed expression, and we will automatically synthesize a martingale and an assertion that holds at termination. We proceed in three stages: extracting a polynomial representing the stochastic process in the program, applying Doob's decomposition to the polynomial representation, and applying optional stopping. We perform symbolic manipulations to simplify the martingale and final fact.

Programs. We consider programs of the form:[2]

$$I; \mathsf{while}\ e\ \mathsf{do}\ (S; B)$$

[1] A basic version of the optional stopping theorem will suffice for our purposes, but there are alternative versions that don't require finite expected stopping time and bounded increments.

[2] We focus on programs for which our method achieves full automation. For instance, we exclude conditional statements because it is difficult to fully automate the resulting simplifications. We note however that there are standard transformations for eliminating conditionals; one such transformation is *if-conversion*, a well-known compiler optimization [1].

where I and B are sequences of deterministic assignments (of the form $\mathcal{X} \leftarrow \mathcal{E}$), and S is a sequence of probabilistic samplings (of the form $\mathcal{S} \xleftarrow{\$} \mathcal{DE}$).

Note that we separate *sample variables* $s \in \mathcal{S}$, which are the target of random samplings, from *process variables* $x \in \mathcal{X}$, which are the target of deterministic assignments. This distinction will be important for our simplifications: we know the moments and correlations of sample variables, while we have less information for process variables. We require that programs assign to sample variables before assigning to process variables in each loop iteration; this restriction is essentially without loss of generality.

We take $\mathcal{DE}$ to be a set of standard distributions over the integers or over finite tuples of integers, to model joint distributions. For instance, we often consider the distribution $\mathsf{Bern}(1/2, \{-1, 1\}) \in \mathcal{DE}$ that returns -1 and $+1$ with equal probability. We assume that all distributions in $\mathcal{DE}$ have bounded support; all moments and correlations of the primitive distributions are finite. We will also assume that distributions do not depend on the program state.

The set $\mathcal{E}$ of expressions is mostly standard, with a few notational conveniences for defining stochastic processes:

$$
\begin{array}{rll}
\mathcal{E} ::= & \mathcal{X} \mid \mathcal{S} \mid \mathcal{X}[-n] & \text{process/sample/history variables} \\
\mid & \mathbb{Z} & \text{constants} \\
\mid & \pi_a(\mathcal{E}) & \text{projections} \\
\mid & \mathcal{E} + \mathcal{E} \mid \mathcal{E} \cdot \mathcal{E} & \text{arithmetic} \\
\mid & \mathcal{E} < \mathcal{E} \mid \mathcal{E} \wedge \mathcal{E} \mid \neg\mathcal{E} & \text{guards}
\end{array}
$$

History variables $\mathcal{X}[-n]$ are indexed by a positive integer n and are used inside loops. The variable $x[-n]$ refers to the value of x assigned n iterations before the current iteration. If the loop has run for fewer than n iterations, $x[-n]$ is specified by the initialization step of our programs:

$$I \triangleq x_1[-n_1] \leftarrow e_1; \cdots ; x_k[-n_k] \leftarrow e_k.$$

Extracting the polynomial. For programs in our fragment, each variable assigned in the loop determines a stochastic process: x is the most recent value, $x[-1]$ is the previous value, etc. In the first stage of our analysis, we extract polynomial representations of each stochastic process from the input program.

We focus on the variables that are mutated in B—each of these variables determines a stochastic process. To keep the notation light, we will explain our process for *first-order* stochastic processes: we only use history variables $x[-1]$ from the past iteration. We will also suppose that there is just one process variable and one sample variable, and only samples from the current iteration are used.

Since our expression language only has addition and multiplication as operators, we can represent the program variable x as a polynomial of other program variables:

$$x = P_x(x[-1], s) \tag{1}$$

Next, we pass to a symbolic representation in terms of (mathematical) random variables. To variable x, we associate the random variable $\{X_i\}_{i \in \mathbb{N}}$ modeling

the corresponding stochastic process, and likewise for the sample variable s. By convention, $i = 0$ corresponds to the initialization step, and $i > 0$ corresponds to the stochastic process during the loop. In other words,

$$x[0] \leftarrow 0; \mathsf{while}\ e\ \mathsf{do}\ s \xleftarrow{\$} d; x \leftarrow x[-1] + s$$

desugars to

$$x[0] \leftarrow 0; i \leftarrow 0; \mathsf{while}\ e\ \mathsf{do}\ i \leftarrow i + 1; s \xleftarrow{\$} d; x[i] \leftarrow x[i-1] + s$$

in a language with arrays instead of history variables. Then, the program variable $x[i]$ corresponds to the random variable X_i.

Then, Equation (1) and the initial conditions specified by the command I give an inductive definition for the stochastic process:

$$X_i = P_x(X_{i-1}, S_i). \tag{2}$$

Applying Doob's decomposition. The second stage of our analysis performs Doob's decomposition on the symbolic representation of the process. We know that the seed expression e must be a polynomial, so we can form the associated stochastic process $\{E_i\}_{i \in \mathbb{N}}$ by replacing program variables by their associated random variable:

$$E_i = P_e(X_i, S_i). \tag{3}$$

(Recall that the initial conditions X_0 and S_0, which define E_0, are specified by the initialization portion I of the program.)

Then, Doob's decomposition produces the martingale:

$$M_i = \begin{cases} E_0 & : i = 0 \\ E_0 + \sum_{j=1}^{i} E_j - \mathbf{E}[E_j \mid \mathcal{F}_{j-1}] & : i > 0. \end{cases}$$

To simplify the conditional expectation, we unfold E_j via Equation (3) and unroll the processes X_i by one step with Equation (2).

Now, we apply our simplification rules; we present a selection in Figure 1. The rules are divided into three groups (from top): linearity of expectation, conditional independence, and distribution information. The first two groups reflect basic facts about expectations and conditional expectations. The last group encodes the moments and correlations of the primitive distributions. We can pre-compute these quantities for each primitive distribution d and store the results in a table.

By the form of Equation (3), the simplification removes all expectations and we can give an explicit definition for the martingale:

$$M_i = \begin{cases} E_0 & : i = 0 \\ Q_e(X_{i-1}, \ldots, X_0, S_i, \ldots, S_1) & : i > 0, \end{cases} \tag{4}$$

where Q_e is a polynomial.

$$\mathbf{E}[c \cdot f + c' \cdot g \mid -] \mapsto c \cdot \mathbf{E}[f \mid -] + c' \cdot \mathbf{E}[g \mid -]$$

$$\mathbf{E}[X_{i-n} \cdot f \mid \mathcal{F}_{i-1}] \mapsto X_{i-n} \cdot \mathbf{E}[f \mid \mathcal{F}_{i-1}] \qquad (n > 0)$$

$$\mathbf{E}[S_i \cdot S'_i \mid \mathcal{F}_{i-1}] \mapsto \mathbf{E}[S_i \mid \mathcal{F}_{i-1}] \cdot \mathbf{E}[S'_i \mid \mathcal{F}_{i-1}] \qquad (S \neq S')$$

$$\mathbf{E}[S_i^k \mid \mathcal{F}_{i-1}] \mapsto G(d)_k \qquad (S \sim d)$$

$$\mathbf{E}[\pi_a(S_i)^p \cdot \pi_b(S_i)^q \mid \mathcal{F}_{i-1}] \mapsto G(d_{a,b})_{p,q} \qquad (S \sim d)$$

Fig. 1: Selection of simplification rules

Applying optional stopping. With the martingale in hand, the last stage of our analysis applies the optional stopping theorem. To meet the technical conditions of the theorem, we need two properties of the loop:

- The expected number of iterations must be finite.
- The martingale must have bounded increments.

These side conditions are non-probabilistic assertions that can already be handled using existing techniques. For instance, the first condition follows from the existence of a *bounded variant* [19]: an integer expression v such that

- $0 \leq v < K$;
- $v = 0$ implies the guard is false; and
- the probability that v decreases is strictly bigger than ϵ

throughout the loop, for ϵ and K positive constants. However in general, finding a bounded variant may be difficult; proving finite expected stopping time is an open area of research which we do not address here.

The second condition is also easy to check. For one possible approach, one can replace stochastic sampling by non-deterministic choice over the support of the distribution, and verify that the seed expression e is bounded using standard techniques [13, 28, 14]. This suffices to show that the martingale M_i has bounded increments. To see why, suppose that the seed expression is always bounded by a constant C. By Doob's decomposition, we have

$$\begin{aligned} |M_i - M_{i-1}| &= \left| \left(\sum_{j=1}^{i} E_j - \mathbf{E}[E_j \mid \mathcal{F}_{j-1}] \right) - \left(\sum_{j=1}^{i-1} E_j - \mathbf{E}[E_j \mid \mathcal{F}_{j-1}] \right) \right| \\ &= |E_i - \mathbf{E}[E_i \mid \mathcal{F}_{j-1}]| \leq 2C, \end{aligned}$$

so the martingale has bounded increments.

Thus, we can apply the optional stopping theorem to Equation (4) to conclude:

$$E_0 = \mathbf{E}[M_0] = \mathbf{E}[M_\tau] = \mathbf{E}[Q_e(X_{\tau-1}, \ldots, X_0, S_\tau, \ldots, S_1)]$$

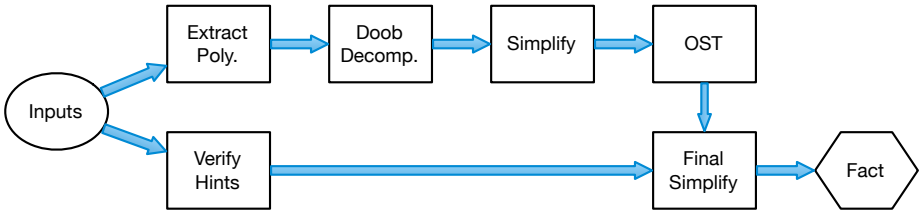

Fig. 2: Tool pipeline

Unlike the simplification step after applying Doob's decomposition, we may not be able to eliminate all expected values. For instance, there may be expected values of X at times before τ. However, if we have additional invariants about the loop, we can often simplify the fact with basic symbolic transformations.

Implementation. We have implemented our process in a Python prototype using the `sympy` library for handling polynomial and summation manipulations [20]. Figure 2 shows the entire pipeline. There are three parts of the input: the program describing a stochastic process, the seed expression, and hint facts. The output is a probabilistic formula that holds at termination.

The most challenging part of our analysis is the last stage: applying OST. First, we need to meet the side conditions of the optional stopping theorem: finite expected iteration count and bounded increments. Our prototype does not verify these side conditions automatically since available termination tools are either not fully automatic [17] or can only synthesize linear ranking supermartingales [6, 9] that are insufficient for the majority of our case studies[3]. Furthermore, the final fact typically cannot be simplified without some basic information about the program state at loop termination. We include this information as *hints*. Hints are first-order formulae over the program variables and the loop counter (represented by the special variable t), and are used as auxiliary facts during the final simplification step. Hints can be verified using standard program verification tools since they are non-probabilistic. In our examples, we manually translate the hints and the program into the input language of EasyCrypt [4], and perform the verifications there. Automating the translation poses no technical difficulty and is left for future work.

We note that the performance of the tool is perfectly reasonable for the examples considered in the next section. For instance, it handles the "ABRACADABRA" example in less than 2 seconds on a modern laptop.

[3] Although most of the ranking supermatingales needed in our case studies are non-linear, the bounded variants are always linear.

4 Examples

Now, we demonstrate our approach on several classic examples of stochastic processes. In each case, we describe the code, the seed expression, and any hints needed for our tool to automatically derive the final simplified fact.

Geometric distribution. As a first application, we consider a program that generates a draw from the geometric distribution by running a sequence of coin flips.

```
x[0] ← 0;
while (z ≠ 0) do
    z ←$ Bern(p, {1, 0});
    x ← x[-1] + z;
end
```

Here, `Bern(p, {1, 0})` is the distribution that returns 1 with probability $p > 0$, and 0 otherwise. The program simply keeps drawing until we sample 0, storing the number of times we sample 1 in x.

We wish to apply our technique to the seed expression x. First, we can extract the polynomial equation:

$$X_i = X_{i-1} + Z_i.$$

Applying Doob's decomposition, our tool constructs and reduces the martingale:

$$M_i = \begin{cases} X_0 & : i = 0 \\ X_i - p \cdot i & : i > 0. \end{cases}$$

To apply optional stopping, we first need to check that the stopping time τ is integrable. This follows by taking z as a bounded variant—it remains in $\{0, 1\}$ and decreases with with probability $p > 0$. Also, the martingale M_i has bounded increments: $|M_i - M_{i-1}|$ should be bounded by a constant. But this is clear since we can use a loop invariant to show that $|X_i - X_{i-1}| \leq 1$, and the increment is

$$|M_i - M_{i-1}| = |X_i - X_{i-1} - p| \leq |X_i - X_{i-1}| + p \leq 1 + p.$$

So, optional stopping shows that

$$0 = \mathbf{E}[X_\tau - p \cdot \tau].$$

With the hint $x = t - 1$—which holds at termination—our tool replaces X_τ by $\tau - 1$ and automatically derives the expected running time:

$$\begin{aligned} 0 &= \mathbf{E}[\tau - 1 - p \cdot \tau] \\ \mathbf{E}[\tau] &= 1/(1 - p). \end{aligned}$$

Gambler's ruin. Our second example is the classic *Gambler's ruin* process. The program models a game where a player starts with $a > 0$ dollars and keeps tossing a fair coin. The player wins one dollar for each head and loses one dollar for each tail. The game ends either when the player runs out of money, or reaches his target of $b > a$ dollars. We can encode this process as follows:

```
x[0] ← a;
while (0 < x < b) do
    z ←$ Bern(1/2, {-1, 1});
    x ← x + z;
end
```

We will synthesize two different martingales from this program, which will yield complementary information once we apply optional stopping. For our first martingale, we use x as the seed expression. Our tool synthesizes the martingale

$$M_i = \begin{cases} X_0 & : i = 0 \\ X_i & : i > 0. \end{cases}$$

So in fact, x is already a martingale.

To apply optional stopping, we first note that x is a bounded variant: it remains in $(0, b)$ and decreases with probability $1/2$ at each iteration. Since the seed expression x is bounded, the martingale M_i has bounded increments. Thus, optional stopping yields

$$a = \mathbf{E}[X_0] = \mathbf{E}[X_\tau].$$

If we give the hint

$$(x = 0) \vee (x = b) \tag{5}$$

at termination, our prototype automatically derives

$$\begin{aligned} a &= \mathbf{E}[X_\tau \cdot \mathbb{1}_{\{X_\tau = 0 \vee X_\tau = b\}}] \\ &= \mathbf{E}[X_\tau \cdot \mathbb{1}_{\{X_\tau = 0\}}] + \mathbf{E}[X_\tau \cdot \mathbb{1}_{\{X_\tau = b\}}] \\ &= 0 \cdot \Pr[X_\tau = 0] + b \cdot \Pr[X_\tau = b] = b \cdot \Pr[X_\tau = b], \end{aligned}$$

so the probability of exiting at b is exactly a/b.

Now, let us take a look at a different martingale generated by the seed expression x^2. Our prototype synthesizes the following martingale:

$$M'_i = \begin{cases} X_0^2 & : i = 0 \\ X_i^2 - i & : i > 0 \end{cases}$$

Again, we can apply optional stopping: x is a bounded variant, and the seed expression x^2 remains bounded in $(0, b^2)$. So, we get

$$a^2 = \mathbf{E}[M_0] = \mathbf{E}[X_\tau^2 - \tau].$$

By using the same hint Equation (5), our prototype automatically derives

$$\begin{aligned} a^2 &= \mathbf{E}[X_\tau^2 \cdot \mathbb{1}_{\{X_\tau = 0 \vee X_\tau = b\}}] - \mathbf{E}[\tau] \\ &= \mathbf{E}[X_\tau^2 \cdot \mathbb{1}_{\{X_\tau = 0\}}] + \mathbf{E}[X_\tau^2 \cdot \mathbb{1}_{\{X_\tau = b\}}] - \mathbf{E}[\tau] \\ &= 0 \cdot \Pr[X_\tau = 0] + b^2 \cdot \Pr[X_\tau = b] - \mathbf{E}[\tau] = b^2 \cdot \Pr[X_\tau = b] - \mathbf{E}[\tau]. \end{aligned}$$

Since we already know that $\Pr[X_\tau = b] = a/b$ from the first martingale $\{M_i\}_{i \in \mathbb{N}}$, this implies that the expected running time of the Gambler's ruin process is

$$\mathbf{E}[\tau] = a(b - a).$$

Gambler's ruin with momentum. Our techniques extend naturally to stochastic processes that depend on variables beyond the previous iteration. To demonstrate, we'll consider a variant of Gambler's ruin process with momentum: besides just the coin flip, the gambler will also gain profit equal to the difference between the *previous two* dollar amounts. Concretely, we consider the following process:

```
x[0] ← a;
x[1] ← a;
while (0 < x < b) do
  z ←$ Bern(p, {-1, 1});
  x ← x[-1] + (x[-1] - x[-2]) + z;
end
```

Note that we must now provide the initial conditions for two steps, since the process is second-order recurrent. Given seed expression x, our tool synthesizes the following martingale:

$$M_i = \begin{cases} X_0 & : i = 0 \\ X_0 + X_i - X_{i-1} & : i > 0 \end{cases}$$

Identical to the Gambler's ruin process, we can verify the side conditions and apply optional stopping, yielding

$$a = \mathbf{E}[M_0] = \mathbf{E}[M_\tau] = \mathbf{E}[X_0 + X_\tau - X_{\tau-1}].$$

Unfolding $X_0 = a$ and simplifying, our tool derives the fact

$$\mathbf{E}[X_\tau] = \mathbf{E}[X_{\tau-1}].$$

We are not aware of existing techniques that can prove this kind of fact—reasoning about the expected value of a variable in the iteration *just prior* to termination.

Abracadabra. Our final example is a classic application of martingale reasoning. In this process, a monkey randomly selects a character at each time step, stopping when he has typed a special string, say "ABRACADABRA". We model this process as follows:

```
match_0[0] ← 1;
match_1[0] ← 0;
...
match_11[0] ← 0;
while (match_11 == 0) do
    s ←$ UnifMatches;
    match_11 ← match_10[-1] * π_11(s);
    match_10 ← match_9[-1] * π_10(s);
    ...
    match_1 ←  match_0[-1] * π_1(s);
end
```

Here, `UnifMatches` is a distribution over tuples that represents a uniform c draw from the letters, where the kth entry is 1 if the c matches the kth word and 0 if not. The variables $\texttt{match}_i$ record whether the i most recent letters match the first i letters of target word; $\texttt{match}_0$ is always 1, since we always match 0 letters.

Now, we will apply Doob's decomposition. Letting L be the number of possible letters and taking the seed expression

$$e \triangleq 1 + L \cdot \texttt{match}_1 + \cdots + L^{11} \cdot \texttt{match}_{11},$$

our tool synthesizes the following martingale:

$$M_i = \begin{cases} 1 + L \cdot X_i^{(1)} + \cdots + L^{11} \cdot X_i^{(11)} & : i = 0 \\ \sum_{j=1}^{i} \left(L \cdot X_j^{(1)} + \cdots + L^{11} \cdot X_j^{(11)} - L^0 \cdot X_{j-1}^{(0)} + \cdots + L^{10} \cdot X_{j-1}^{(10)} \right) & : i > 0, \end{cases}$$

where $X^{(j)}$ is the stochastic process corresponding to $\texttt{match}_j$. The dependence on L is from the expectations of projections of `UnifMatches`, which are each $1/L$—the probability of a uniformly random letter matching any fixed letter.

To apply the optional stopping theorem, note that the seed expression e is bounded in $(0, L^{12})$, and $1 + L + \cdots + L^{11} - e$ serves as a bounded variant: take the highest index j such that $\texttt{match}_j = 1$, and there is probability $1/L$ that we increase the match to get $\texttt{match}_{j+1} = 1$, decreasing the variant. So, we have

$$1 = \mathbf{E}[M_0] = \sum_{j=1}^{\tau} \mathbf{E}[L \cdot X_j^{(1)} + \cdots + L^{11} \cdot X_j^{(11)}] - \mathbf{E}[L^0 \cdot X_{j-1}^{(0)} + \cdots + L^{10} \cdot X_{j-1}^{(10)}].$$

Our tool simplifies and uses the hints $X_j^{(11)} = 0$ and $X_j^{(0)} = 1$ for $j < \tau$ to derive

$$1 = L^0 \cdot \mathbf{E}[X_\tau^{(0)}] + \cdots + L^{11} \cdot \mathbf{E}[X_\tau^{(11)}] - \mathbf{E}[\tau].$$

For the target string "ABRACADABRA", we use hints

$$\begin{aligned} (\texttt{match}_{11} = 1) &\implies (\texttt{match}_4 = 1) \\ (\texttt{match}_{11} = 1) &\implies (\texttt{match}_1 = 1) \\ (\texttt{match}_{11} = 1) &\implies (\texttt{match}_0 = 1) \\ (\texttt{match}_{11} = 1) &\implies (\texttt{match}_j = 0). \qquad (\text{for } j \neq 0, 1, 4, 11) \end{aligned}$$

For example, if $\texttt{match}_{11}$ is set then the full string is matching "ABRACADABRA", so the most recently seen four characters are "ABRA". This matches the first four letters of "ABRACADABRA", so $\texttt{match}_4$ is also set. The hint can be proved from a standard loop invariant.

Our tool derives the expected running time:

$$\mathbf{E}[\tau] = L^1 + L^4 + L^{11}.$$

Benchmarks.

To give an idea of the efficiency of our tool, we present some benchmarks for our examples in Table 1. We measured timing on a recent laptop with a 2.4 GHz Intel Core processor with 8 GB of RAM. We did not optimize for the performance; we expect that the running time could be greatly improved with some tuning.

Example	Running time (s)
GEOM	0.14
GAMBLE	0.11
GAMBLE2	0.17
MINIABRA	0.87
FULLABRA	3.58

Table 1: Preliminary benchmarks.

The example MINIABRA is a smaller version of the ABRACADABRA example, where the alphabet is just $\{0, 1\}$, and we stop when we sample the sequence 111; FULLABRA is the full ABRACADABRA example.

While there is a growing body of work related to martingale techniques for program analysis (see the following section), it is not obvious how to compare benchmarks. Existing work focuses on searching for martingale expressions within some search space; this is a rather different challenge than synthesizing a single martingale from a programmer-provided seed expression. In particular, if the seed expression happens to already be a martingale by some lucky guess, our tool will simply return the seed expression after checking that it is indeed a martingale.

5 Related work

Martingales. Martingale theory is a central tool in fields like statistics, applied mathematics, control theory, and finance. When analyzing randomized algorithms, martingales can show tight bounds on tail events [30]. In the verification community, martingales are used as invariant arguments, and as variants arguments to prove almost sure termination [5, 6, 18, 9]. Recently, martingale approaches were extended to prove more complex properties. Chakarov, Voronin,

and Sankaranarayanan [8] propose proof rules for proving persistence and recurrence properties. Dimitrova, Ferrer Fioriti, Hermanns, and Majumdar [16] develop a deductive proof system for PCTL*, with proof rules based on martingales and supermartingales.

Probabilistic Hoare logic. McIver and Morgan [27] propose a Hoare-like logic that is quite similar to our approach of using martingales and OST. Their approach is based on *weakest pre-expectations*, which are an extension of Dijkstra's weakest preconditions [15] based on "backward" conditional expectations. Their probabilistic invariants are similar to submartingales, as the expected value of the invariant at the beginning of the execution lower bounds the expected value of the invariant at termination. Their proof rule also requires an additional constraint to ensure soundness, but it requires a limiting argument that is more difficult to automate compared to our bounded increment condition. We could relax our condition using a weaker version of OST that generalizes their condition [33]. Another substantial difference with our approach is that their logic supports non-deterministic choices—ours does not. It is not obvious how we can extend our synthesis approach to the non-probabilistic case as we heavily rely on the concept of filtration, not applicable in the presence of non-determinism.

Probabilistic model checking. In the last twenty years, model checking technology for probabilistic models have made impressive strides [26, 11, 23] (Baier and Katoen [3] provide overview). The main advantage of model checking is that it requires nothing from the user; our technique requires a seed expression. However, model checking techniques suffer from the state explosion problem—the time and memory consumption of the model checking algorithm depends on the number of reachable states of the program. Our approach can be used to verify infinite and parametric programs without any performance penalty, as we work purely symbolically. For example, a probabilistic model checker can find the expected running time of the gambler's ruin process for concrete values of a and b but they cannot deduce the solution for the general case, unlike our technique.

Invariant synthesis. There are several approaches for synthesizing probabilistic invariants. Katoen et al. [22] propose the first complete method for the synthesis of McIver and Morgan's probabilistic linear invariants. It is an extension of the constraint solving approach by Colón, Sankaranarayanan, and Sipma [12] for the synthesis of (non-probabilistic) linear invariants. Chakarov and Sankaranarayanan [6] later extended this work to martingales and ranking supermartingales. Chakarov and Sankaranarayanan [7] propose a new notion of probabilistic invariants that generalizes the notion of supermatingales. They give a synthesis approach based on abstract interpretation, but it is not clear how their techniques can prove properties at termination time. Chen et al. [10] propose a synthesis tool for verifying Hoare triples in the McIver and Morgan logic, using a combination of Lagrange's interpolation, sampling, and counterexample guided search. One of the novelties is that they can synthesize non-linear invariants. The main disadvantages is that one must manually check the soundness condition,

and one must provide a pre-expectation. For instance, we can apply the method of Chen et al. [10] to the gambler's ruin process only if we already know that the expected running time is $a(b-a)$. In contrast, we can deduce $\mathbf{E}[\tau] = a(b-a)$ knowing only that $\mathbf{E}[\tau]$ is finite.

Expected running time. As the termination time of a probabilistic program is a random quantity, it is natural to measure its performance using the average running time. Rough bounds can be obtained from martingale-based termination proofs [18]. Recently, Chatterjee et al. [9] showed that arbitrary approximations can be obtained from such proofs when the program is linear. They use Azuma's inequality to obtain a tail distribution of the running time, and later they model check a finite unrolling of the loop. Monniaux [29] propose a similar approach that uses abstract interpretation to obtain the tail distribution of the running time. Kaminski, Katoen, Matheja, and Olmedo [21] extend Nielson's proof system [31] to bound the expected running time of probabilistic programs.

Recurrence analysis. Our synthesis approach is similar to the use of recurrences relations for the synthesis of non-probabilistic invariants [2, 32, 24]. The main idea is to find syntactic or semantic recurrences relations, and later simplify them using known closed forms to obtain loop invariants. In essence, we apply algebraic identities to simplify the complex martingales from Doob's decomposition. The difference is that our simplifications are more complex as we cannot always obtain a closed form but a simpler summation. However, we obtain the same closed form when we apply Doob's decomposition to inductive variables. Another difference is that we rely on the syntactic criteria to identify which values are predictable and which values are random.

6 Conclusion

We proposed a novel method for automatically synthesizing martingales expressions for stochastic processes. The basic idea is to transform any initial expression supplied by the user into a martingale using Doob's decomposition theorem. Our method complements the state-of-the-art synthesis approaches based on constraint solving. On one hand, we always output a martingale expression, we are able to synthesize non-inductive martingales, and since we do not rely on quantifier elimination, we can synthesize polynomial expression of very high degree. On the other hand, we do not provide any completeness result, and the shape of martingale is difficult to predict.

We considered several classical case studies from the literature, combining our synthesis method with the optional stopping theorem and non-probabilistic invariants to infer properties at termination time in a fully automatic fashion.

Future work includes extending our approach to programs with arrays and improving the tool with automated procedures for checking side-conditions. It would also be interesting to consider richer programs, say distributions with parameters that depend on program state. Another possible direction would

be improving the simplification procedures; possibly, the tool could produce simpler facts. Experimenting with more advanced computer-algebra systems and designing simplification heuristics specialized to handling the conditional expectations synthesized by Doob's decomposition are both promising future directions. It would also be interesting to integrate our method as a special tool in systems for interactive reasoning about probabilistic computations.

Acknowledgments. We thank the anonymous reviewers for their helpful comments. This work was partially supported by NSF grants TWC-1513694 and CNS-1065060, and by a grant from the Simons Foundation (#360368 to Justin Hsu).

Bibliography

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, pages 177–189, New York, NY, USA, 1983. ACM. ISBN 0-89791-090-7.

[2] Z. Ammarguellat and W. L. Harrison III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), White Plains, New York*, pages 283–295, 1990.

[3] C. Baier and J. Katoen. *Principles of model checking*. MIT Press, 2008. ISBN 978-0-262-02649-9.

[4] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin. Computer-aided security proofs for the working cryptographer. In *IACR International Cryptology Conference (CRYPTO), Santa Barbara, California*, pages 71–90, 2011.

[5] O. Bournez and F. Garnier. Proving positive almost-sure termination. In *International Conference on Rewriting Techniques and Applications (RTA), Nara, Japan*, pages 323–337, 2005.

[6] A. Chakarov and S. Sankaranarayanan. Probabilistic program analysis with martingales. In *International Conference on Computer Aided Verification (CAV), Saint Petersburg, Russia*, pages 511–526, 2013.

[7] A. Chakarov and S. Sankaranarayanan. Expectation invariants for probabilistic program loops as fixed points. In *International Symposium on Static Analysis (SAS), Munich, Germany*, pages 85–100, 2014.

[8] A. Chakarov, Y.-L. Voronin, and S. Sankaranarayanan. Deductive proof for almost sure persistence and recurrence properties. In M. Chechik and J. Raskin, editors, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Eindhoven, The Netherlands*, volume 9636 of *Lecture Notes in Computer Science*, pages 260–279. Springer, 2016.

[9] K. Chatterjee, H. Fu, P. Novotný, and R. Hasheminezhad. Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. pages 327–342, 2016.

[10] Y. Chen, C. Hong, B. Wang, and L. Zhang. Counterexample-guided polynomial loop invariant generation by lagrange interpolation. In *International Conference on Computer Aided Verification (CAV), San Francisco, California*, pages 658–674, 2015.

[11] F. Ciesinski and C. Baier. LiQuor: A tool for qualitative and quantitative linear time analysis of reactive systems. In *Third International Conference on the Quantitative Evaluation of Systems (QEST), Riverside, California*, pages 131–132, 2006.

[12] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *International Conference on Computer Aided Verification (CAV), Boulder, Colorado*, pages 420–432, 2003.

[13] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM Symposium on Principles of Programming Languages (POPL), Tucson, Arizona*, pages 84–96, 1978.

[14] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Budapest, Hungary*, pages 337–340, 2008.

[15] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[16] R. Dimitrova, L. M. Ferrer Fioriti, H. Hermanns, and R. Majumdar. Probabilistic CTL*: The deductive way. In M. Chechik and J. Raskin, editors, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Eindhoven, The Netherlands*, volume 9636 of *Lecture Notes in Computer Science*, pages 280–296. Springer, 2016.

[17] J. Esparza, A. Gaiser, and S. Kiefer. Proving termination of probabilistic programs using patterns. In *International Conference on Computer Aided Verification (CAV), Berkeley, California*, pages 123–138, 2012.

[18] L. M. Ferrer Fioriti and H. Hermanns. Probabilistic termination: Soundness, completeness, and compositionality. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Mumbai, India*, pages 489–501, 2015.

[19] S. Hart, M. Sharir, and A. Pnueli. Termination of probabilistic concurrent program. *ACM Transactions on Programming Languages and Systems*, 5(3): 356–380, 1983.

[20] D. Joyner, O. Čertík, A. Meurer, and B. E. Granger. Open source computer algebra systems: SymPy. *ACM Commun. Comput. Algebra*, 45(3–4):225–234, Jan. 2012.

[21] B. Kaminski, J.-P. Katoen, C. Matheja, and F. Olmedo. Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs. In *European Symposium on Programming (ESOP), Eindhoven, The Netherlands*, Jan. 2016.

[22] J. Katoen, A. McIver, L. Meinicke, and C. C. Morgan. Linear-invariant generation for probabilistic programs: Automated support for proof-based methods. In *International Symposium on Static Analysis (SAS), Perpignan, France*, pages 390–406, 2010.

[23] J. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen. The ins and outs of the probabilistic model checker MRMC. *Perform. Eval.*, 68 (2):90–104, 2011.

[24] L. Kovács. Reasoning algebraically about p-solvable loops. In C. R. Ramakrishnan and J. Rehof, editors, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Budapest, Hungary*, volume 4963 of *Lecture Notes in Computer Science*, pages 249–264. Springer, 2008.

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

[26] M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *International Conference on Computer Aided Verification (CAV), Snowbird, Utah*, pages 585–591, 2011.

[27] A. McIver and C. Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer, New York, 2005.

[28] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[29] D. Monniaux. An abstract analysis of the probabilistic termination of programs. In *International Symposium on Static Analysis (SAS), Paris, France*, pages 111–126, 2001.

[30] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995. ISBN 0-521-47465-5.

[31] H. R. Nielson. A hoare-like proof system for analysing the computation time of programs. *Sci. Comput. Program.*, 9(2):107–136, 1987.

[32] E. Rodríguez-Carbonell and D. Kapur. Generating all polynomial invariants in simple loops. *J. Symb. Comput.*, 42(4):443–476, 2007.

[33] D. Williams. *Probability with Martingales*. Cambridge University Press, 1991.