

# Formal Certification of Randomized Algorithms

## Abstract

Randomized algorithms have broad applications throughout computer science. They also pose a challenge for formal verification: even intuitive properties of simple programs can have elaborate proofs, mixing program verification with probabilistic reasoning.

We present Ellora, a tool-assisted framework for the interactive verification of general properties of randomized algorithms. The central component of Ellora is a new and expressive program logic for imperative programs with adversarial code. In particular, the logic supports fine-grained reasoning about probabilistic loops by offering different reasoning principles according to the termination behavior, and assertions can model key notions from probability theory, such as probabilistic independence and expected values. We show soundness for the logic and develop an implementation on top of the EasyCrypt proof assistant.

We demonstrate the strength of Ellora in two ways. First, we embed several specialized logics into Ellora: an adaptation of greatest pre-expectation calculus from Kozen [26], Morgan et al. [31] (restricted to loop-free programs without non-determinism), the union bound logic from Barthe et al. [6], and a novel Hoare logic for reasoning about distribution laws and probabilistic independence. Second, we formally verify several classical randomized algorithms.

## 1. Introduction

Randomized algorithms are fundamental objects of study in theoretical computer science, with broad applications to computational fields like cryptography and machine learning. They also present a challenging target for formal verification. Often presented as imperative programs, they satisfy appealing properties such as computational efficiency and various notions of probabilistic accuracy. While the properties often capture intuitive features, their correctness can be subtle—even simple properties may require intricate proofs using complex mathematical theorems.

While mathematical results can play a role in proving correctness for all programs, probabilistic or not, proofs for randomized algorithms frequently apply tools from a broad collection of concepts and results from probability theory, like distribution laws, probabilistic independence, and concentration bounds. As a consequence, a formal framework for reasoning about randomized algorithms should provide mechanisms for smoothly applying these common tools, as they are traditionally used in paper-and-pen proofs.

For deductive verification, where specifications are given by a pre-condition and a post-condition, a natural verification strategy is to let assertions be interpreted as sets of distributions over program states. This idea was first proposed by Ramshaw [33], and subsequently refined by other researchers [8, 13, 34]. However, existing systems have several shortcomings. First, typical examples of randomized algorithms and their properties are difficult to express. Existing program logics do not support assertions about general expected values, a fundamental part of many target properties, and restrict sampling to Boolean distributions. All other distributions, like the uniform distribution or the normal distribution, need to be simulated with loops.

Second, existing reasoning principles for proving specifications are also limited. Prior work does not consider reasoning about lossy programs, i.e. programs which terminate with probability strictly less than 1, and about programs with adversarial code, a natural concept in many applications from security and privacy (which naturally carry a notion of adversary), and from game theory and mechanism design (where adversaries model strategic agents). Furthermore, proofs commonly require low-level reasoning about the semantics of programs and assertions. For instance, reasoning about loops involves semantic side conditions that can be difficult to prove; many program logics use non-standard logical connectives to reason about random sampling and conditionals.

## The Ellora framework

In this paper we introduce Ellora, a mechanized framework for general-purpose, interactive reasoning about randomized algorithms. The central component is a new probabilistic program logic alleviating the shortcomings of previous work in several respects; we highlight the main novelties here.

*Reasoning about loops.* Proving a property of a probabilistic loop typically requires analyzing its termination behavior and establishing a loop invariant. Moreover, the class of loop invariants that can be soundly used depends on the termination behavior. We identify three classes of assertions that can be used for reasoning about probabilistic loops, and provide a proof rule for each one:

- arbitrary assertions for *certainly terminating* loops, i.e. loops that terminate in a finite amount of iterations;
- *topologically closed* assertions for *almost surely* terminating loops, i.e. loops terminating with probability 1;
- *downwards closed* assertions for arbitrary loops.

Our definition of topologically closed assertion is reminiscent of Ramshaw [33]; the stronger notion of downwards closed assertion appears to be new.

Besides broadening the class of loops that can be analyzed, a diverse collection of rules can lead to simpler proofs. For instance, if the loop is certainly terminating, then there is no need to prove semantic side conditions. Likewise, there is no need to consider the termination behavior of the loop when the invariant is downwards and topologically closed. For example, in many applications, especially cryptography, the target property is that a “bad” event has low probability:  $\Pr[E] \leq k$ . This assertion is downwards and topologically closed, so it can be a sound loop invariant regardless of the termination behavior.

*Reasoning about adversaries.* *Adversaries* are special procedures, specified by an interface listing the concrete procedures that an adversary can call, along with restrictions like how many calls an adversary may make. Adversaries are widely used in cryptography, where security notions are described using experiments in which one or several adversaries interact with a challenger, and in game theory and mechanism design, where they are used for modeling strategic agents. Adversaries can also model *online* algorithms, where an external party interacts with an algorithm.

*Reasoning with specialized tools.* The first central goal of Ellora is providing support for specialized reasoning principles from existing, “paper” proofs of randomized algorithms. These patterns do not always apply, but they are lightweight methods to prove specific types of commonly-used properties. By simplifying and organizing proofs about common properties, these patterns form an indispensable part of a toolkit for reasoning about randomized algorithms. We demonstrate support in Ellora for these principles by embedding several specialized logics that neatly capture specific aspects of probabilistic reasoning: the *union bound* logic by Barthe et al. [6] (for proving upper bounds on probabilities), the loop-free fragment of the *greatest pre-expectation calculus* due to McIver and Morgan [28] (for computing expectations), and an *independence and distribution law* logic. This last logic is new and potentially of independent interest, designed to reason about independence in a lightweight way that is common in paper proofs.

*Reasoning about probabilistic notions.* The second central goal of Ellora is general reasoning about common properties and notions from existing proofs, like probabilities, expected values, distribution laws and probabilistic independence. There is prior work covering some of these aspects—most notably for expected values—but it remains practically challenging to carry out general probabilistic arguments within a unified system. We demonstrate Ellora on a collection of case studies, including textbook examples and a randomized routing algorithm. Our experience suggests that Ellora is capable of both expressing and reasoning about the

toolbox of properties found in existing proofs, freely applying theorems from probability theory.

*Implementation.* We develop a full-featured implementation of our framework on top of EasyCrypt, a general-purpose proof assistant for reasoning about probabilistic programs. Assertions in our implementation have a concrete syntax, encoding a two-level assertion language. The first level contains state predicates—deterministic assertions about a single memory—while the second layer includes probabilistic assertions constructed from probabilities and expected values over discrete distributions. While the concrete language cannot express arbitrary predicates on distributions, it is a natural fit for all properties and invariants of randomized algorithms that we have encountered. More importantly, the assertion language supports several syntactic tools to simplify verification:

- an automated procedure for generating pre-conditions of non-looping commands, inspired from prior work on greatest pre-expectations [26, 31]; and
- syntactic conditions for the closedness and termination properties required for soundness of the loop rules, and syntactic conditions for soundness of the frame and adversary rules.

In order to carry out full verification of example algorithms in our implementation, we also develop a partial formalization of probability theory in Ellora, including common tools like concentration bounds (e.g., the Chernoff bound), Markov’s inequality, and theorems about probabilistic independence.

## Contributions

To summarize, we present the following contributions.

- A probabilistic Hoare logic with general predicates on distributions, rules for handling different kinds of probabilistically terminating loops and procedure calls, and a mechanized proof of soundness for the logic;
- a concrete version of the logic with an assertion language suitable for syntactic tools, and an implementation within a general-purpose theorem prover;
- embeddings of three specialized reasoning tools: a core version of the greatest pre-expectation calculus from Morgan et al. [31], the union bound logic from Barthe et al. [6], and a novel Hoare logic for reasoning about distribution laws and probabilistic independence; and
- case studies demonstrating formal verification of randomized algorithms.

*Comparison with expectation-based techniques.* To date, arguably the most mature systems for deductive verification of randomized algorithms are derived from *expectation-based* techniques. These systems consider *expectations*, functions  $E$  from states to real numbers; the name comes by considering a program as mapping an input state  $s$  to a distribution  $\mu(s)$  on

output states, when the expected value of  $E$  on  $\mu(s)$  is an expectation. Roughly speaking, expectation-based approaches offer deductive principles to compositionally apply the effect of the program to the expectation, transforming it until it can be analyzed purely mathematically. A probabilistic property, expressed as a formula involving probabilities and expectations, is proved by separately transforming each component in this way. Classical examples include PDDL [26] and pGCL [31]; the latter work also considers non-determinism. Expectation-based systems are very elegant and have a neat meta-theory. Moreover, they have been used for verifying several randomized algorithms. In particular, there have been efforts to mechanize these systems and to verify sophisticated case studies, e.g. Hurd [18], Hurd et al. [20].

A direct comparison with probabilistic Hoare logics is difficult, since the two approaches are quite different. In broad strokes, program logics can verify richer properties in one shot, have assertions that are easier to understand, and can make assertions about the input viewed as a distribution, while expectation-based approaches can transform expectations mechanically and reason about loops without semantic side conditions. (Vásquez et al. [39] provide a more thorough comparison.) By incorporating tools inspired by expectation-based techniques and designing loop rules with syntactic side conditions, we aim for the best of both worlds within Ellora.

## 2. Example: accuracy of private sum

```

proc psum (a[N]:int array):
  var s:int, l[N]:int array;
  s ← 0;
  for j ← 0 to N-1 do
    l[j]  $\stackrel{\$}{\leftarrow}$   $\mathcal{L}_\epsilon$ ;
    s ← s + l[j] + a[j];
  return s

```

**Figure 1.** Private sum

We illustrate the style of reasoning that we want to capture in Ellora using a simple program inspired by differential privacy. The program (Fig. 1) computes the private sum of an array as follows: it

draws samples from the Laplace distribution  $\mathcal{L}_\epsilon$  for each element, and then adds each element with its noise to the current sum.

Our goal is to establish an accuracy bound for private sum: the return value  $s$ , which holds the sum of  $a$ , should be close to the true sum  $\widehat{s}$  of  $a$  with high probability. First, we establish the following loop invariant for each iteration  $j$ :

$$\left( s - \sum_{k=0}^j a[k] = \sum_{k=0}^j \iota[k] \right) \wedge \#(\iota[0], \dots, \iota[j]) \wedge \bigwedge_{1 \leq i \leq j} \iota[i] \sim \mathcal{L}_\epsilon$$

where the second conjunct states that  $\iota[0], \dots, \iota[j]$  are independent random variables and the third conjunct states that they are all distributed according to  $\mathcal{L}_\epsilon$ . Next, we use the second and third conjuncts of the invariant to apply a *concentration bound*. This theorem gives a formula  $T : (0, 1) \rightarrow \mathbb{R}$  such that for every failure probability  $b \in (0, 1)$ ,  $T(b)$  upper

bounds the sum of  $\iota$  except with probability  $b$ :

$$\Pr [|s - \widehat{s}| > T(b)] = \Pr \left[ \left| \sum_{k=0}^{N-1} \iota[k] \right| > T(b) \right] \leq b$$

This simple example highlights two desirable features of a proof system for probabilistic programs:

1. the ability to both prove and use properties of distributions, like probabilistic independence and i.i.d. variables;
2. the ability to internalize basic theorems of probability, like concentration bounds.

To compare, using the concentration bound yields a better than the bound than simpler approaches like aHL [6], a lightweight program logic which can reason about accuracy of differentially private computations by using the *union bound* but cannot reason about independence. Expectation-based approaches (e.g., PDDL [26] or pGCL [31]) can in principle establish the same bound, but the verification strategy would be rather different from the proof we sketched above. Since there is no direct way to use concepts like probabilistic independence or concentration bounds, expectation-based proofs propagate probabilities like  $\Pr \left[ \left| \sum_{k=0}^{N-1} \iota[k] \right| > T(b) \right]$ , throughout the program. This is difficult to do when there are distributions with infinite support (e.g., the Laplace distribution), or parameters (e.g., the number of samples  $N$ ). Furthermore, this reasoning must be repeated throughout the proof for each application of the concentration bound. By working with predicates on distributions, we are able to directly model probabilistic independence and concentration bounds, giving a more natural and concise formal proof.

## 3. Programs and assertions

**Programs.** We base our development on pWhile, a core language with deterministic assignments, probabilistic assignments, conditionals, loops, procedure calls and an **abort** statement which halts the computation with no result. Probabilistic assignments are of the form  $x \stackrel{\$}{\leftarrow} g$ , which assigns a value sampled according to the distribution  $g$  to the program variable  $x$ . The syntax of statements is defined by the grammar:

$$s ::= \mathbf{skip} \mid \mathbf{abort} \mid x \leftarrow e \mid x \stackrel{\$}{\leftarrow} g \mid s; s \\ \mid \mathbf{if } e \mathbf{ then } s \mathbf{ else } s \mid \mathbf{while } e \mathbf{ do } s \mid x \leftarrow \mathcal{F}(e) \mid x \leftarrow \mathcal{A}(e)$$

where  $x, e$ , and  $g$  range over (typed) variables in  $\mathcal{X}$ , expressions in  $\mathcal{E}$  and distribution expressions in  $\mathcal{D}$  respectively.  $\mathcal{E}$  is defined inductively from  $\mathcal{X}$  and a set  $\mathcal{F}$  of simply typed function symbols, while  $\mathcal{D}$  is defined by combining a set of distribution symbols  $\mathcal{S}$  with expressions in  $\mathcal{E}$ . For instance,  $e_1 + e_2$  is a valid expression, and **Bern**( $e$ )—the Bernoulli distribution with parameter  $e$ —is a valid distribution expression. We assume that expressions, distribution expressions, and statements are typed in the usual way with **SDist**( $T$ )

the type for probability (sub-)distributions over the type  $T$ . Ellora can be flexibly extended with custom functions and types.

We distinguish two kinds of procedure calls:  $\mathcal{A}$  is a set of external procedure names, and  $\mathcal{F}$  is a set of internal procedure names. We assume we have access to the code of internal procedures, but not the code of external procedures. We think of external procedures as controlled by some external *adversary*, who can select the next input in an interactive algorithm.

The denotational semantics of programs is adapted from the seminal work of Kozen [25] and interprets programs as sub-distribution transformers. We first define memories as type-preserving mappings from variables to values, and let  $\mathbf{State}$  denote the set of memories. The semantics of a statement  $s$  w.r.t. to some sub-distribution  $\mu$  over memories is another sub-distribution over memories, denoted  $\llbracket s \rrbracket_\mu$ . When  $s$  contains adversarial procedures, the semantics is further parametrized by an interpretation of the adversary calls. We defer details to the supplemental materials.

We conclude this section with a taxonomy of the termination behavior of statements and loops.

**Definition 1** (Lossless). *A (closed) statement  $s$  is lossless iff for every sub-distribution  $\mu$ ,  $|\llbracket s \rrbracket_\mu| = |\mu|$ , where  $|\mu|$  is the total probability of  $\mu$ .*

Programs that are not lossless are called *lossy*.

**Definition 2** (Certain and almost sure termination). *A loop `while  $b$  do  $s$`  is:*

- certainly (c.) terminating *if there exists  $n$  such that for every sub-distribution  $\mu$ :  $|\llbracket \mathbf{while } b \mathbf{ do } s \rrbracket_\mu| = |\llbracket (\mathbf{if } b \mathbf{ then } s)^n \rrbracket_\mu|$ .*
- almost surely (a.s.) terminating *if it is lossless.*

Certain termination is similar to termination in deterministic programs, whereas almost sure termination is probabilistic in nature: the program always terminates eventually, but we may not be able to give a single finite bound for all executions since particular executions may proceed arbitrarily long. Note that certain termination need not entail losslessness.

**Assertions.** We model assertions as predicates on states.

**Definition 3** (Assertions and satisfaction). *The set  $\mathbf{Assn}$  of assertions is defined as  $\mathcal{P}(\mathbf{SDist}(\mathbf{State}))$ . We write  $\eta(\mu)$  for  $\mu \in \eta$ .*

Usual set operations are lifted to assertions using their logical counterparts, e.g.,  $\eta \wedge \eta' \triangleq \eta \cap \eta'$  and  $\neg\eta \triangleq \bar{\eta}$ . Beside these standard constructions, we frequently use the following assertions. Given a predicate  $\phi$  over states, we let  $\Box\phi$  be the assertion defined by:

$$\Box\phi \triangleq \lambda\mu. \forall m. m \in \text{supp}(\mu) \implies \phi(m)$$

where  $\text{supp}(\mu)$  is the set of all memories with non-zero probability under  $\mu$ . Intuitively, this means that  $\phi$  holds

on all memories that we may sample from the distribution. Moreover, given two assertions  $\eta_1$  and  $\eta_2$ , we let  $\eta_1 \oplus \eta_2$  be the assertion defined by the clause:

$$\eta_1 \oplus \eta_2 \triangleq \lambda\mu. \exists \mu_1, \mu_2. \mu = \mu_1 + \mu_2 \wedge \eta_1(\mu_1) \wedge \eta_2(\mu_2)$$

The sub-distribution  $\mu_1 + \mu_2$  is the sub-distribution that has probabilities given by the sum (see, e.g. Kozen [25]). Intuitively, this assertion means that the sub-distribution is the sum of two sub-distributions, such that  $\eta_1$  holds on the first piece and  $\eta_2$  holds on the second piece. Finally, given an assertion  $\eta$  and a function  $F$  from  $\mathbf{SDist}(\mathbf{State})$  to  $\mathbf{SDist}(\mathbf{State})$ , we let  $\eta[F]$  be the assertion defined by the clause:  $\eta[F] \triangleq \lambda\mu. \eta(F(\mu))$ .

Now, we can define the closedness properties of assertions. These properties will later be used to achieve soundness of the rules for **while** loops.

**Definition 4** (Closedness properties).

- An assertion  $\eta$  is *t-closed* if for every converging sequence of sub-distributions  $(\mu_n)_{n \in \mathbb{N}}$  such that  $\eta(\mu_n)$  for all  $n \in \mathbb{N}$  then

$$\eta\left(\lim_{n \rightarrow \infty} \mu\right).$$

- An assertion  $\eta$  is *d-closed* if it is *t-closed* and downward closed, that is for every sub-distributions  $\mu \leq \mu'$ ,  $\eta(\mu')$  implies  $\eta(\mu)$ .

While closedness is a semantic property, there are several sufficient conditions that are easier to check. First, both *t-closed* and *d-closed* assertions are closed under finite boolean combinations, universal quantification over arbitrary sets and existential quantification over finite sets. We can give some examples:

- assertions of the form  $p_1 \bowtie p_2$ , where  $\bowtie$  is a non-strict comparison operator ( $\bowtie \in \{\leq, \geq, =\}$ ) for bounded probabilistic expressions  $p_1, p_2$ —for instance probabilities or expectations of bounded variables—are *t-closed*. There are simple examples where *t-closedness* fails for unbounded expressions. An example of a *t-closed* assertion is the equivalence of two variables  $x, y$ :

$$\overbrace{\forall n \in \mathbb{N}}^{\text{Universal quantification}}, \quad \overbrace{\Pr[x = n]}^{\text{Bounded expression}} = \overbrace{\Pr[y = n]}^{\text{Bounded expression}}$$

- assertions of the form  $p_1 \leq k$  for bounded probabilistic expression  $p_1$  are *d-closed*.

## 4. Proof system

In this section, we introduce a program logic for proving properties of probabilistic programs, and prove its soundness.

**Judgments and proof rules.** Judgments are of the form  $\{\eta\} s \{\eta'\}$ , where  $\eta, \eta' \in \mathbf{Assn}$ .

**Definition 5.** *A judgment  $\{\eta\} s \{\eta'\}$  is valid, written  $\models \{\eta\} s \{\eta'\}$ , if  $\eta'(\llbracket s \rrbracket_\mu)$  for every probabilistic state  $\mu$  such that  $\eta(\mu)$ .*

$\frac{\text{CONSEQ}}{\eta_0 \Rightarrow \eta_1 \quad \frac{\{ \eta_1 \} s \{ \eta_2 \} \quad \eta_2 \Rightarrow \eta_3}{\{ \eta_0 \} s \{ \eta_3 \}}}$		
$\frac{\text{ABORT}}{\{ \eta \} \mathbf{abort} \{ \square \perp \}}$	$\frac{\text{ASSGN}}{\eta' \triangleq \eta[[x \leftarrow e]] \quad \{ \eta' \} x \leftarrow e \{ \eta \}}$	$\frac{\text{SKIP}}{\{ \eta \} \mathbf{skip} \{ \eta \}}$
$\frac{\text{SAMPLE}}{\eta' \triangleq \eta[[x \stackrel{s}{\leftarrow} g]] \quad \{ \eta' \} x \stackrel{s}{\leftarrow} g \{ \eta \}}$	$\frac{\text{SEQ}}{\frac{\{ \eta_0 \} s_1 \{ \eta_1 \} \quad \{ \eta_1 \} s_2 \{ \eta_2 \}}{\{ \eta_0 \} s_1 ; s_2 \{ \eta_2 \}}}$	
$\frac{\text{COND}}{\frac{\{ \eta_1 \wedge \square e \} s_1 \{ \eta'_1 \} \quad \{ \eta_2 \wedge \square \neg e \} s_2 \{ \eta'_2 \}}{\{ (\eta_1 \wedge \square e) \oplus (\eta_2 \wedge \square \neg e) \} \mathbf{if} \ e \ \mathbf{then} \ s_1 \ \mathbf{else} \ s_2 \{ \eta'_1 \oplus \eta'_2 \}}}$		
$\frac{\text{SPLIT}}{\frac{\{ \eta_1 \} s \{ \eta'_1 \} \quad \{ \eta_2 \} s \{ \eta'_2 \}}{\{ \eta_1 \oplus \eta_2 \} s \{ \eta'_1 \oplus \eta'_2 \}}}$		
$\frac{\text{CALL}}{\frac{\{ \eta \} f_{\mathbf{arg}} \leftarrow e ; f_{\mathbf{init}} ; f_{\mathbf{body}} \{ \eta'[[x \leftarrow f_{\mathbf{res}}]] \}}{\{ \eta \} x \leftarrow f(e) \{ \eta' \}}}$		
$\frac{\text{FRAME}}{\frac{\text{separated}(\eta, \text{mod}(s)) \quad s \text{ is lossless}}{\{ \eta \} s \{ \eta \}}}$		

**Figure 2.** Structural and basic rules

Figure 2 describes the structural and basic rules of the proof system. Validity of judgments is preserved under standard structural rules, like the rule of consequence [CONSEQ]. The rule of consequence is especially important for our purposes, since it serves as the interface between the program logic and theorems probability theory. For instance, this rule is how we can apply concentration bounds and other mathematical results.

The rules for **skip**, assignments, random samplings and sequences are all straightforward. The rule for **abort** requires  $\square \perp$  to hold after execution; this assertion uniquely characterizes the resulting null sub-distribution. The rules for assignments and random samplings are semantical; we give more concrete versions in the next section. The [CALL] rule for procedure calls reduces to proving the given pre- and post-conditions on the body of the procedure.

The rule [COND] for conditionals is unusual in that the post-condition must be of the form  $\eta_1 \oplus \eta_2$ ; this reflects the semantics of conditionals, which splits the initial probabilistic state depending on the guard, runs both branches, and adds the resulting two probabilistic states.

The next two rules ([SPLIT] and [FRAME]) are critical for local reasoning. The [SPLIT] rule reflects the additivity of the

semantics and can be used for simultaneously recombining pre- and post-conditions using the  $\oplus$  operator. The [FRAME] rule states that lossless statements preserve assertions that are not influenced by its set  $\text{mod}(s)$  of *modified* variables: the variables on the left of an assignment, a random sampling or a procedure call. In this setting, we say that an assertion  $\eta$  is *separated* from a set of variables  $X$ , written  $\text{separated}(\eta, X)$ , if  $\eta(\mu_1) \iff \eta(\mu_2)$  for any distributions  $\mu_1, \mu_2$  s.t.  $|\mu_1| = |\mu_2|$  and  $\mu_1|_{\bar{X}} = \mu_2|_{\bar{X}}$  where for a set  $S$ ,  $\mu|_S$  is defined as:

$$\mu|_S : m \in \mathbf{State}|_S \mapsto \Pr_{m' \sim \mu} [m = m'|_S]$$

Intuitively, an assertion is separated from a set of variables  $X$  if every two sub-distributions that agree on the variables outside  $X$  either both satisfy the assertion, or both refute the assertion.

Figure 3 presents the rules for **while** loops and external procedure calls. The [WHILE] rule has three instantiations, depending on the termination behavior of the loop. As usual, we must provide a loop invariant; in our case, a loop invariant is an arbitrary assertion that is preserved by one (guarded) iteration of the loop. The instantiations consider arbitrary, almost surely, and certainly terminating loops. In the general case, when no restriction are required about the termination behavior, we require the invariant to be  $d$ -closed. This condition can be weakened for terminating loops: in the case of a loop that terminates surely, a  $t$ -closed invariant is sufficient; whereas we do not require anything for loops terminating certainly.

The rule [ADV] for external procedure calls follows the same idea: it states when an assertion that is preserved by oracle calls is also preserved by the external procedure call. Some framing conditions are required, similar to the ones of the [FRAME] rule: the invariant must not be influenced by the state writable by the external procedures, which also must be lossless. Similar to the loop rule, different flavors exist. In Figure 3, we only give the most general one where the invariant is required to be  $d$ -closed. However, for example, this restriction can be removed by bounding the number of calls the external procedure can make to oracles, leading to a rule akin to the certain termination case of the loop rule.

**Soundness.** Our proof system is sound w.r.t. the semantics.

**Theorem 6 (Soundness).** *Every judgment  $\{ \eta \} s \{ \eta' \}$  provable using the rules of our logic is valid.*

Completeness of the logic is left for future work.

## 5. A two-level syntax for assertions

So far, we have seen a version of Ellora where assertions are arbitrary predicates on distributions. While this version is quite general, it is desirable for practical applications to define a syntax for assertions and to develop specialized proof rules which are easier to use.

$$\begin{array}{c}
\text{WHILE-X} \\
\frac{\{\eta\} \text{ if } b \text{ then } s \{\eta\} \quad \mathcal{C}_X}{\{\eta\} \text{ while } b \text{ do } s \{\eta \wedge \Box \neg b\}} \\
\text{ADV} \\
\frac{\text{separated}(\eta, \{x, \mathfrak{s}\}) \quad \eta \text{ is } d\text{-closed} \quad a \text{ is lossless}}{\forall f \in a_{\text{ocl}}, x \in \mathcal{X}_a^{\mathcal{E}}, e \in \mathcal{E}. \{\eta\} x \leftarrow f(e) \{\eta\}} \\
\{\eta\} x \leftarrow a(e) \{\eta\}
\end{array}$$

SIDE CONDITIONS:

$$\begin{array}{ll}
\mathcal{C}_{\text{CTerm}} \triangleq \forall \mu. \eta(\mu) \implies \exists k \in \mathbb{N}. \Pr_{m \sim \nu} [\llbracket b \rrbracket_m] = 0 & \text{where } \nu \triangleq \llbracket (\text{if } b \text{ then } s) \rrbracket_{\mu} \\
\mathcal{C}_{\text{ASTerm}} \triangleq \eta \text{ } t\text{-closed and } \forall \mu. \eta(\mu) \implies |\nu| = 1 & \text{where } \nu \triangleq \llbracket \text{while } b \text{ do } s \rrbracket_{\mu} \\
\mathcal{C}_{\text{ATerm}} \triangleq \eta \text{ } d\text{-closed} &
\end{array}$$

**Figure 3.** Rules for while loops and external calls

**Assertions.** Fig. 4 presents a two-level assertion language which is sufficiently expressive for typical assertions that arise in the analysis of randomized algorithms. The assertion language supports probabilistic assertions and state assertions. A *probabilistic assertion*  $\eta$  is a formula built from comparison of probabilistic expressions, using first-order quantifiers and connectives, and the special connective  $\oplus$ . A *probabilistic expression*  $p$  can be a logical variable  $v$ , an operator applied to probabilistic expressions  $o(\vec{p})$  (constants are 0-ary operators), or the expectation  $\mathbb{E}[\tilde{e}]$  of a state expression  $\tilde{e}$ . A *state expression*  $\tilde{e}$  is either a program variable  $x$ , the characteristic function  $\mathbf{1}_{\phi}$  of a state assertion  $\phi$ , an operator applied to state expressions  $o(\vec{\tilde{e}})$ , or the expectation  $\mathbb{E}_{v \sim g}[\tilde{e}]$  of state expression  $\tilde{e}$  in a given distribution  $g$ . Finally, a *state assertion*  $\phi$  is a first-order formula over program variables. Note that the set of operators is left unspecified but we assume that all the expressions in  $\mathcal{E}$  and  $\mathcal{D}$  can be encoded by operators.

The interpretation of the concrete syntax is as expected. The interpretation of probabilistic assertions is relative to a valuation  $\rho$  which maps logical variables to values, and is an element of  $\text{Assn}$ . The definition of the interpretation is straightforward; the only interesting case is  $\llbracket \mathbb{E}[\tilde{e}] \rrbracket_{\mu}^{\rho}$  which is defined by  $\mathbb{E}_{m \sim \mu} [\llbracket \tilde{e} \rrbracket_m^{\rho}]$ , where  $\llbracket \tilde{e} \rrbracket_m^{\rho}$  is the interpretation of the state expression  $\tilde{e}$  in the memory  $m$  and valuation  $\rho$ . The interpretation of state expressions is a mapping from memories to values, which can be lifted to a mapping from distributions over memories to distributions over values. The definition of the interpretation is straightforward; the most interesting case is for expectation  $\llbracket \mathbb{E}_{v \sim g}[\tilde{e}] \rrbracket_m^{\rho} \triangleq \mathbb{E}_{w \sim \llbracket g \rrbracket_m^{\rho}} [\llbracket \tilde{e} \rrbracket_m^{\rho[v:=w]}]$ . We present the full interpretations in the supplemental materials.

Many standard concepts from probability theory have a natural representation in our syntax. For example:

- the probability that  $\phi$  holds in some probabilistic state is represented by the probabilistic expression  $\Pr[\phi] \triangleq \mathbb{E}[\mathbf{1}_{\phi}]$ ;
- probabilistic independence of state expressions  $\tilde{e}_1, \dots, \tilde{e}_n$  is modeled by the probabilistic assertion  $\#\{\tilde{e}_1, \dots, \tilde{e}_n\}$ ,

$$\begin{array}{ll}
\tilde{e} ::= x \mid v \mid \mathbf{1}_{\phi} \mid \mathbb{E}_{v \sim g}[\tilde{e}] \mid o(\vec{\tilde{e}}) & \text{(S-expr.)} \\
\phi ::= \tilde{e} \bowtie \tilde{e} \mid FO(\phi) & \text{(S-assn.)} \\
p ::= v \mid o(\vec{p}) \mid \mathbb{E}[\tilde{e}] & \text{(P-expr.)} \\
\eta ::= p \bowtie p \mid \eta \oplus \eta \mid FO(\eta) & \text{(P-assn.)} \\
\bowtie \in \{=, <, \leq\} \quad o \in Ops & \text{(Operators)}
\end{array}$$

**Figure 4.** Assertion syntax

defined by the clause<sup>1</sup>

$$\forall v_1 \dots v_n, \Pr[\top]^{n-1} \Pr\left[\bigwedge_{i=1 \dots n} \tilde{e}_i = v_i\right] = \prod_{i=1 \dots n} \Pr[\tilde{e}_i = v_i];$$

- losslessness of a distribution is modeled by the probabilistic assertion  $\mathcal{L} \triangleq \Pr[\top] = 1$ ;
- a state expression  $\tilde{e}$  distributed according to a law  $g$  is modeled by the probabilistic assertion  $\tilde{e} \sim g$  defined as:

$$\forall w, \Pr[\tilde{e} = w] = \mathbb{E}[\mathbb{E}_{v \sim g}[\mathbf{1}_{v=w}]].$$

The inner expectation computes the probability that  $v$  drawn from  $g$  is equal to a fixed  $w$ ; the outer expectation weights the inner probability by the probability of each value of  $w$ .

We can easily define  $\Box$  operator from the previous section in our new syntax:  $\Box \phi \triangleq \Pr[\neg \phi] = 0$ .

**Syntactic proof rules.** Now that we have a concrete syntax for assertions, we can give syntactic versions of many of the existing proof rules. Such proof rules are often easier to use, since they avoid reasoning about the semantics of commands and assertions. We tackle the non-looping rules first, beginning with the following syntactic rules for assignment and sampling:

<sup>1</sup> The term  $\Pr[\top]^{n-1}$  is necessary since we work with sub-distributions; for distributions,  $\Pr[\top] = 1$  and we recover the usual definition.

$$\begin{array}{c} \text{SAMPLE} \\ \hline \{\eta[x := e]\} x \leftarrow e \{\eta\} \end{array} \qquad \begin{array}{c} \text{ASSGN} \\ \hline \{\mathcal{P}_x^g(\eta)\} x \stackrel{g}{\leftarrow} \{\eta\} \end{array}$$

The rule for assignment is the usual rule from Hoare logic, replacing the program variable  $x$  by its corresponding expression  $e$  in the pre-condition. The replacement  $\eta[x := e]$  is done recursively on the probabilistic assertion  $\eta$ ; for expectation it is defined by

$$\mathbb{E}[\tilde{e}[x := e]] \triangleq \mathbb{E}[\tilde{e}[x := e]]$$

where  $\tilde{e}[x := e]$  is the syntactic substitution of the variable  $x$  by the expression  $e$  in  $\tilde{e}$ .

The rule for sampling is a generalization of assignment using a probabilistic substitution operator  $\mathcal{P}_x^g(\eta)$ , which replaces all occurrences of  $x$  in  $\eta$  by a new integration variable  $t$  and records that  $t$  is drawn from  $g$ . More formally,  $\mathcal{P}_x^g(\eta)$  is defined recursively on probabilistic assertions; for expectation, it is defined as

$$\mathcal{P}_x^g(\mathbb{E}[\tilde{e}]) \triangleq \mathbb{E}[\mathbb{E}_{t \sim g}[\tilde{e}[x := t]]].$$

Next, we turn to the loop rule. The side conditions from Fig. 3 are purely semantic, while in practice it is more convenient to use a sufficient condition in the Hoare logic. We give sufficient conditions for ensuring certain and almost-sure termination in Fig. 5.

The first side condition  $\mathcal{C}_{\text{CTerm}}$  shows certain termination given a strictly decreasing *variant*  $\tilde{e}$  that is bounded below, similar to how a decreasing variant shows termination for deterministic programs. The second side condition  $\mathcal{C}_{\text{ASTerm}}$  shows almost-sure termination given a probabilistic variant  $\tilde{e}$ , which must be bounded both above and below. While  $\tilde{e}$  may increase with some probability, it must decrease with strictly positive probability. This sufficient condition for almost-sure termination was previously considered by Hart et al. [17] for probabilistic transition systems, and also used in expectation-based approaches [19, 30].

**Precondition calculus.** With a concrete syntax for assertions, we are also able to incorporate syntactic reasoning principles. One classic tool is Morgan and McIver’s *greatest pre-expectation*, which we take as inspiration for a precondition calculus for the loop-free fragment of Ellora. Given an assertion  $\eta$  and a loop-free statement  $s$ , we wish to mechanically construct an assertion  $\eta^*$  that is the pre-condition of  $s$  that implies  $\eta$  as a post-condition.

Given a statement  $s$  and a probabilistic assertion  $\eta$ , the computation of the pre-condition replaces each expectation expression  $p$  inside  $\eta$  by an expression  $p^*$  that has the same denotation before running  $s$  as  $p$  after running  $s$ . This process yields an assertion  $\eta^*$  that, interpreted before running  $s$ , is logically equivalent to  $\eta$  interpreted after running  $s$ .

The computation rules for pre-conditions are defined in Fig. 6. For a probability assertion  $\eta$ , its pre-condition

$\text{pc}(s, \eta)$  corresponds to  $\eta$  where the expectation expressions of the form  $\mathbb{E}[\tilde{e}]$  are replaced by their corresponding *preterm*,  $\text{pe}(s, \mathbb{E}[\tilde{e}])$ . Preterms correspond loosely to Morgan and McIver’s *pre-expectations*—we will make this correspondence more precise in the next section. The main interesting cases for computing preterms are for random sampling and conditionals. For random sampling the result is  $\mathcal{P}_x^g(\mathbb{E}[\tilde{e}])$ , which corresponds to the [SAMPLE] rule. For conditionals, the expectation expression is split into a part where  $e$  is true and a part where  $e$  is not true. We restrict the expectation to a part satisfying  $e$  with the following operator:

$$\mathbb{E}[\tilde{e}]|_e \triangleq \mathbb{E}[\tilde{e} \cdot \mathbf{1}_e]$$

This corresponds to the expected value of  $\tilde{e}$  on the portion of the distribution where  $e$  is true.

Then, we can build the pre-condition calculus into Ellora.

**Theorem 7.** *Let  $s$  be a non-looping command. Then, the following rule is derivable in the concrete version of Ellora:*

$$\text{PC} \frac{}{\{\text{pc}(s, \eta)\} s \{\eta\}}$$

## 6. Embedding logics

While our presentation of Ellora so far is suitable for general-purpose reasoning about probabilistic programs, in practice proofs typically use more lightweight, specific reasoning principles to prove certain assertions. To see that such patterns can also be naturally used in Ellora, we consider embeddings of three tools in our framework: the union bound logic from Barthe et al. [6], a fragment of pGCL [31], and a new logic for reasoning about independence.

**Union bound logic.** Barthe et al. [6] have recently introduced a lightweight program logic, called aHL, for estimating accuracy of randomized computations. One main application of aHL is proving accuracy of randomized algorithms, both in the offline and online settings—i.e. with adversary calls. aHL is based on the union bound, a basic tool from probability theory, and has judgments of the form

$$\models_{\beta} \{\Phi\} s \{\Psi\},$$

where  $s$  is a statement,  $\Phi$  and  $\Psi$  are first-order formulae over program variables, and  $\beta$  is a probability, i.e.  $\beta \in [0, 1]$ . A judgment  $\models_{\beta} \{\Phi\} s \{\Psi\}$  is valid if for every memory  $m$  such that  $\Phi(m)$ , the probability of  $\neg\Psi$  in  $\llbracket s \rrbracket_m$  is upper bounded by  $\beta$ , i.e.  $\text{Pr}_{\llbracket s \rrbracket_m}[\neg\Psi] \leq \beta$ .

Figure 7 presents some key rules of aHL, including a rule for sampling from the Laplace distribution  $\mathcal{L}_{\epsilon}$  centered around  $e$ . The predicate  $\mathcal{C}_{\text{CTerm}}(k)$  indicates that the loop terminates in at most  $k$  steps on any memory that satisfies the pre-condition. Moreover,  $\beta$  is a function of  $\epsilon$ . aHL has a simple embedding into Ellora.

**Theorem 8 (Embedding of aHL).** *If  $\models_{\beta} \{\Phi\} s \{\Psi\}$  is derivable, then  $\{\square\Phi\} s \{\mathbb{E}[\mathbf{1}_{\neg\Psi}] \leq \beta\}$  is derivable in Ellora.*

$$\begin{array}{l}
\mathcal{C}_{\text{CTerm}} \triangleq \{ \mathcal{L} \wedge \Box(\tilde{e} = k \wedge 0 < k \wedge b) \} s \{ \mathcal{L} \wedge \Box(\tilde{e} < k) \} \quad \tilde{e} : \mathbb{N} \\
\quad \quad \quad \models \eta \Rightarrow (\exists \dot{y}. \Box \tilde{e} \leq \dot{y}) \wedge \Box(\tilde{e} = 0 \Rightarrow \neg b) \\
\mathcal{C}_{\text{ASTerm}} \triangleq \{ \mathcal{L} \wedge \Box(\tilde{e} = k \wedge 0 < k \leq K \wedge b) \} s \{ \mathcal{L} \wedge \Box(0 \leq \tilde{e} \leq K) \wedge \Pr[\tilde{e} < k] \geq \epsilon \} \quad \tilde{e} : \mathbb{N} \\
\quad \quad \quad \models \eta \Rightarrow \Box(0 \leq \tilde{e} \leq K \wedge \tilde{e} = 0 \Rightarrow \neg b) \\
\quad \quad \quad \models \eta \text{ } t\text{-closed}
\end{array}$$

**Figure 5.** Side-conditions for loop rules

$$\begin{array}{l}
\text{pe}(s_1; s_2, \mathbb{E}[\tilde{e}]) \triangleq \text{pe}(s_1, \text{pe}(s_2, \mathbb{E}[\tilde{e}])) \\
\text{pe}(x \leftarrow e, \mathbb{E}[\tilde{e}]) \triangleq \mathbb{E}[\tilde{e}][x := e] \\
\text{pe}(x \stackrel{s}{\leftarrow} g, \mathbb{E}[\tilde{e}]) \triangleq \mathcal{P}_x^g(\mathbb{E}[\tilde{e}]) \\
\text{pe}(\text{if } e \text{ then } s_1 \text{ else } s_2, \mathbb{E}[\tilde{e}]) \triangleq \text{pe}(s_1, \mathbb{E}[\tilde{e}])|_e + \text{pe}(s_2, \mathbb{E}[\tilde{e}])|_{\neg e} \\
\text{pc}(s, p_1 \bowtie p_2) \triangleq \text{pc}(s, p_1) \bowtie \text{pc}(s, p_2)
\end{array}$$

$\text{gpe}(s, E)(m) = \mathbb{E}_{s' \sim \mu(m)}[E]$

**Figure 6.** Precondition calculus (selected)

$$\begin{array}{c}
\frac{}{\models_{\beta} \{ \top \} x \stackrel{s}{\leftarrow} \mathcal{L}_{\epsilon}(e) \{ |x - e| \leq \frac{1}{\epsilon} \log \frac{1}{\beta} \}} \\
\frac{\models_{\beta_1} \{ \Phi \} s_1 \{ \Theta \} \quad \models_{\beta_2} \{ \Theta \} s_2 \{ \Psi \}}{\models_{\beta_1 + \beta_2} \{ \Phi \} s_1; s_2 \{ \Psi \}} \\
\frac{\models_{\beta} \{ \Phi \} c \{ \Phi \} \quad \mathcal{C}_{\text{CTerm}}(k)}{\models_{k \cdot \beta} \{ \Phi \} \text{ while } e \text{ do } c \{ \Phi \wedge \neg e \}}
\end{array}$$

**Figure 7.** aHL proof rules (selected)

Of course, there are valid judgments that cannot be derived with the aHL rules but are provable using Ellora. For instance, the proof of the private sum example from § 2 requires applying concentration bounds and reasoning about independence, which are not supported in aHL.

**The logic pGCL.** Probabilistic Guarded Command Language (pGCL) [31] is a well-studied deductive system for reasoning about programs. This language is the same as the core imperative language that we consider, except instead of a random sampling command from general distributions  $x \stackrel{s}{\leftarrow} g$ , pGCL encodes random choice with a probabilistic guarded command of the form  $s_1 \oplus_p s_2$ . This command executes  $s_1$  with probability  $p$ , otherwise it executes  $s_2$ . For the embedding, we will simulate this command by sampling from the Bernoulli (coin flip) distribution with parameter  $p$ , denoted  $B(p)$ . pGCL also notably supports reasoning about various kinds of non-deterministic choice; we do not consider these features here. We also will only embed loop-free pGCL commands.

The key object in pGCL reasoning is an *expectation*—a map from states to real numbers—and the key tool is *greatest pre-expectation*. This procedure takes a command  $s$  and an

expectation  $E$ , and mechanically computes an expectation  $\text{gpe}(s, E)$  such that if we view  $s$  as a map from an input memory  $m$  to an output distribution  $\mu(m)$ , then

$$\text{gpe}(s, E)(m) = \mathbb{E}_{s' \sim \mu(m)}[E]$$

for every memory  $m$ . That is, the greatest pre-expectation takes an expectation  $E$  on the *output* distribution, and transforms it into an expectation  $E'$  that takes the same value as  $E$  when  $E'$  is evaluated on the *input* memory. In this way, we can calculate a target final expectation by propagating it backwards through the program, until we compute a mathematical formula for the expectation as a function of the input memory only.

For the embedding, we will assume that each expectation  $E$  can be directly interpreted as a state expression  $\tilde{e}$ . The embedding uses the precondition calculus presented in the previous section. We will need one technical lemma.

**Lemma 9.** *For every state expression  $\tilde{e}$ , loop-free and deterministic pGCL program  $s$ , and real number  $\alpha$ ,*

$$\Box(\text{gpe}(s, \tilde{e}) = \alpha) \implies \text{pc}(\lceil s \rceil, \mathbb{E}[\tilde{e}] = \alpha)$$

where  $\lceil s \rceil$  is the program obtained from  $s$  by replacing all occurrences of the probabilistic choice  $s_1 \oplus_p s_2$  by  $x \stackrel{s}{\leftarrow} B(p)$ ; **if**  $x$  **then**  $s_1$  **else**  $s_2$  for a fresh variable  $x$ .

Then, we can embed a fragment of pGCL into Ellora.

**Theorem 10** (Embedding of core pGCL). *For every state expression  $\tilde{e}$ , loop-free and deterministic pGCL program  $s$ , and real number  $\alpha$ , the following judgment is derivable:*

$$\{ \Box(\text{gpe}(s, \tilde{e}) = \alpha) \} \lceil s \rceil \{ \mathbb{E}[\tilde{e}] = \alpha \}.$$

*Proof.* Since  $\lceil s \rceil$  is loop-free, we can derive  $\{ \text{pc}(\lceil s \rceil, \mathbb{E}[\tilde{e}] = \alpha) \} \lceil s \rceil \{ \mathbb{E}[\tilde{e}] = \alpha \}$ . by rule [PC]. By Lemma 9 we also have  $\Box(\text{gpe}(s, \tilde{e}) = \alpha) \implies \text{pc}(\lceil s \rceil, \mathbb{E}[\tilde{e}] = \alpha)$ , so we can conclude with the rule [CONSEQ].  $\square$

**Law and Independence Logic.** Our final example is a proof system for reasoning about probabilistic independence and distribution laws. This type of reasoning is common when analyzing randomized algorithms; yet, it is particularly hard to capture formally. Many existing program logics for probabilistic programs either cannot capture these notions or provide poor support. We begin by describing the law and independence logic IL, a proof system with intuitive rules that



are easy to apply and amenable to automation. For simplicity, we only consider programs which sample from the binomial distribution, and have deterministic control flow—for lack of space, we also omit procedure calls.

**Definition 11** (Assertions). *IL assertions have the grammar:*

$$\xi := \det(e) \mid \#E \mid e \sim B(e, p) \mid \top \mid \perp \mid \xi \wedge \xi$$

where  $e \in \mathcal{E}$ ,  $E \subseteq \mathcal{E}$ , and  $p \in [0, 1]$ .

The assertion  $\det(e)$  states that  $e$  is deterministic in the current distribution, i.e., there is at most one element in the support of its interpretation. The assertion  $\#E$  states that the expressions in  $E$  are independent, as formalized in the previous section. The assertion  $e \sim B(m, p)$  states that  $e$  is distributed according to a binomial distribution with parameter  $m$  (where  $m$  can be an expression) and constant probability  $p$ , i.e. the probability that  $e = k$  is equal to the probability that exactly  $k$  independent coin flips return heads using a biased coin that returns heads with probability  $p$ .

Assertions can be seen as an instance of a logical abstract domain, where the order between assertions is given by implication based on a small number of axioms. Examples of such axioms include independence of singletons, irreflexivity of independence, anti-monotonicity of independence, an axiom for the sum of binomial distributions, and rules for deterministic expressions:

$$\begin{aligned} \#\{x\} \quad \#\{x, x\} &\iff \det(x) \quad \#(E \cup E') \implies \#E \\ e \sim B(m, p) \wedge e' \sim B(m', p) \wedge \#\{e, e'\} &\implies \\ e + e' &\sim B(m + m', p) \\ \bigwedge_{1 \leq i \leq n} \det(e_i) &\implies \det(f(e_1, \dots, e_n)) \end{aligned}$$

**Definition 12.** *Judgments of the logic are of the form  $\{\xi\} s \{\xi'\}$ , where  $\xi$  and  $\xi'$  are IL-assertions. A judgment is valid if it is derivable from the rules of Fig. 8 (structural rules and rule for sequential composition are similar to those from § 4 and omitted).*

The rule [IL-ASSGN] for deterministic assignments is as in § 4. The rule [IL-SAMPLE] for random assignments yields as post-condition that the variable  $x$  and a set of expressions  $E$  are independent assuming that  $E$  is independent before the sampling, and moreover that  $x$  follows the law of the distribution that it is sampled from. The rule [IL-COND] for conditionals requires that the guard is deterministic, and that each of the branches satisfies the specification; if the guard is not deterministic, there are simple examples where the rule is not sound. The rule [IL-WHILE] for loops requires that the loop is certainly terminating with a deterministic guard. Note that the requirement of certain termination could be avoided by restricting the structural rules such that a statement  $s$  has deterministic control flow whenever  $\{\xi\} s \{\xi'\}$  is derivable.

$$\text{IL-ASSGN} \frac{}{\{\xi[x := e]\} x \leftarrow e \{\xi\}}$$

$$\text{IL-SAMPLE} \frac{\{x\} \cap \text{FV}(E) \cap \text{FV}(e) = \emptyset}{\{\#E\} x \stackrel{s}{\leftarrow} B(e, p) \{\#(E \cup \{x\}) \wedge x \sim B(e, p)\}}$$

$$\text{IL-SEQ} \frac{\{\xi\} s_1 \{\xi'\} \quad \{\xi'\} s_2 \{\xi''\}}{\{\xi\} s_1; s_2 \{\xi''\}}$$

$$\text{IL-COND} \frac{\{\xi\} s_1 \{\xi'\} \quad \{\xi\} s_2 \{\xi'\} \quad \xi \implies \det(b)}{\{\xi\} \text{ if } b \text{ then } s_1 \text{ else } s_2 \{\xi'\}}$$

$$\text{IL-WHILE} \frac{\{\xi\} s \{\xi\} \quad \xi \implies \det(b) \quad \mathcal{C}_{\text{CTerm}}}{\{\xi\} \text{ while } b \text{ do } s \{\xi\}}$$

**Figure 8.** Selected proof rules of IL logic

We now turn to the embedding. The embedding of IL assertions into general assertions is immediate, except for  $\det(e)$  which is translated as  $\Box e \vee \Box \neg e$ . We let  $\bar{\xi}$  denote the translation of  $\xi$ .

**Theorem 13** (Embedding and soundness of IL logic). *If  $\{\xi\} s \{\xi'\}$  is derivable in the IL logic, then  $\{\bar{\xi}\} s \{\bar{\xi}'\}$  is derivable in (the syntactic variant of) Ellora. As a consequence, every derivable judgment  $\{\xi\} s \{\xi'\}$  is valid.*

*Proof sketch.* By induction on the derivation. The interesting cases are conditionals and loops. For conditionals, the soundness follows from the soundness of the rule:

$$\frac{\{\eta\} s_1 \{\eta'\} \quad \{\eta\} s_2 \{\eta'\} \quad \Box e \vee \Box \neg e}{\{\eta\} \text{ if } e \text{ then } s_1 \text{ else } s_2 \{\eta'\}}$$

To prove the soundness of this rule, we proceed by case analysis on  $\Box e \vee \Box \neg e$ . We treat the case  $\Box e$ ; the other case is similar. In this case,  $\eta$  is equivalent to  $\eta_1 \wedge \Box e \oplus \eta_2 \wedge \Box \neg e$ , where  $\eta_1 = \eta$  and  $\eta_2 = \perp$ . Let  $\eta'_1 = \eta'$  and  $\eta'_2 = \Box \perp$ ; again,  $\eta'_1 \oplus \eta'_2$  is logically equivalent to  $\eta'$ . The soundness of the rule thus follows from the soundness of the [COND] and [CONSEQ] rules. For loops, there exists a natural number  $n$  such that **while**  $b$  **do**  $s$  is semantically equivalent to **(if**  $b$  **then**  $s$ ) <sup>$n$</sup> . By assumption  $\{\xi\} s \{\xi\}$  holds, and thus by induction hypothesis  $\{\bar{\xi}\} s \{\bar{\xi}\}$ . We also have  $\xi \implies \det(b)$ , and hence  $\{\bar{\xi}\} \text{ if } b \text{ then } s \{\bar{\xi}\}$ . We conclude by using the [SEQ] rule.  $\square$

To illustrate our system IL, consider the statement  $s$  in Fig. 9 which flips a fair coin  $N$  times and counts the number of heads. Using the logic, we can prove that  $s \sim B(N \cdot (N + 1)/2, 1/2)$  is a valid post-condition for  $s$ . We

omit the proof that the loop guard is deterministic, and focus on the distribution of  $s$ . We take the following invariant:

$$s \sim B(j(j+1)/2, 1/2)$$

The invariant holds initially, as  $0 \sim B(0, 1/2)$ . For the inductive case, we have to establish

$$\{s \sim B(0, 1/2)\} c_0 \{s \sim B((j+1)(j+2)/2, 1/2)\}$$

where  $c_0$  represents the loop body, i.e.  $x \stackrel{\$}{\leftarrow} B(j, 1/2); s \leftarrow s + x$ . First, we apply the rule for sequence taking as intermediate assertion

$$s \sim B(j(j+1)/2, 1/2) \wedge x \sim B(j, 1/2) \wedge \#\{x, s\}$$

The first premise follows from the rule for random assignment and structural rules. The second premise follows from the rule for deterministic assignment and the rule of consequence, applying axioms about sums of binomial distributions.

```

proc sum () =
  var s:int, x:int;
  s ← 0;
  for j ← 1 to N do
    x  $\stackrel{\$}{\leftarrow}$  B(j, 1/2);
    s ← s + x;
  return s

```

**Figure 9.** Sum of bin.

We briefly comment on several limitations of IL. First, IL is restricted to programs with deterministic control flow, but this restriction could be partially relaxed by enriching IL with assertions for conditional independence. Such assertions are already expressible in the logic of Ellora; adding conditional independence would significantly broaden the scope of the IL proof system and open the possibility to rely on axiomatizations of conditional independence (e.g., based on graphoids [32]). Second, the logic only supports sampling from binomial distributions. It is possible to enrich the language of assertions with clauses  $s \sim g$  where  $g$  can model other distributions, like the uniform distribution or the Laplace distribution. The main design challenge is finding a core set of useful facts about these distributions. Enriching the logic and automating the analysis are interesting avenues for further work.

## 7. Case studies

In this section, we will demonstrate Ellora on a selection of examples; we present further examples in the supplemental material. Together, they exhibit a wide variety of different proof techniques and reasoning principles which are available in the Ellora’s implementation.

**Hypercube routing.** We will begin with the *hypercube routing* algorithm [37, 38]. Consider a network topology (the *hypercube*) where each node is labeled by a bitstring of length  $D$ , and two nodes are connected by an edge if and only if the two corresponding labels differ in exactly one bit position.

In the network, there is initially one packet at each node, and each packet has a unique destination. The algorithm

implements a routing strategy based on *bit fixing*: if the current position has bitstring  $i$ , and the target node has bitstring  $j$ , we compare the bits in  $i$  and  $j$  from left to right, moving along the edge that corrects the first differing bit. Valiant’s algorithm uses randomization to guarantee that the total number of steps grows *logarithmically* in the number of packets. In the first phase, each packet  $i$  select an intermediate destination  $\rho(i)$  uniformly at random, and use bit fixing to reach  $\rho(i)$ . In the second phase, each packet use bit fixing to go from  $\rho(i)$  to the destination  $j$ . We will focus on the first phase, since the reasoning for the second phase is nearly identical. We can model the strategy with the following code, using some syntactic sugar for the **for** loops<sup>2</sup>.

```

proc route (D T : int) :
  var  $\rho$ , pos, usedBy : node map;
  var nextE : edge;
  pos ← Map.init id  $2^D$ ;  $\rho \leftarrow$  Map.empty;
  for  $i \leftarrow 1$  to  $2^D$  do  $\rho[i] \stackrel{\$}{\leftarrow} [1, 2^D]$ 
  for  $t \leftarrow 1$  to  $T$  do
    usedBy ← Map.empty;
    for  $i \leftarrow 1$  to  $2^D$  do
      if  $\text{pos}[i] \neq \rho[i]$  then
        nextE ← getEdge pos[i]  $\rho[i]$ ;
        if usedBy[nextE] =  $\perp$  then
          usedBy[nextE] ←  $i$ ; // Mark edge used
          pos[i] ← dest nextE // Move packet
    return (pos,  $\rho$ )

```

We assume that initially the position of the packet  $i$  is at node  $i$  (see `Map.init`). Then, we initialize the random intermediate destinations  $\rho$ . The remaining loop encodes the evaluation of the routing strategy iterated  $T$  time. The variable `usedBy` is a map that logs if an edge is already used by a packet, it is empty at the beginning of each iteration. For each packet, we try to move it across one edge along the path to its intermediate destination. The function `getEdge` returns the next edge to follow, following the bit-fixing scheme. If the packet can progress (its edge is not used), then its current position is updated and the edge is marked as used.

We show that if the number of timesteps  $T$  is  $4D + 1$ , then all packets reach their intermediate destination in at most  $T$  steps, except with a small probability  $2^{-2D}$  of failure. That is, the number of timesteps grows linearly in  $D$ , logarithmic in the number of packets. This is formalized in our system as:

$$\{T = 4D + 1\} \text{ route } \{\text{Pr}[\exists i. \text{pos}[i] \neq \rho[i]] \leq 2^{-2D}\}$$

**Modeling infinite processes.** Our second example is the *coupon collector* process. The algorithm draws a uniformly random coupon (we have  $N$  coupon) on each day, terminating when it has drawn at least one of each kind of coupon. The code of the algorithm is displayed in Figure 10. The code uses the array `cp` to keep track of the coupons seen so far; `t` to keep track of the number of steps taken before seeing a new coupon; `X` to keep track of the total number of steps.

<sup>2</sup>We recall that the number of node in a hypercube of dimension  $D$  is  $2^D$  so each node can be identified by a number in  $[1, 2^D]$ .

Our goal is to bound the average number of iterations. This is formalized in our logic as:

$$\{\mathcal{L}\} \text{ coupon } \left\{ \mathbb{E}[X] = \sum_{i \in [1, N]} \left( \frac{N}{N-i+1} \right) \right\}.$$

```

proc coupon (N : int) :
  var int cp[N], t[N];
  var int X ← 0;
  for p ← 1 to N do
    ct ← 0;
    cur  $\stackrel{s}{\leftarrow}$  [1, N];
    while cp[cur] = 1 do
      ct ← ct + 1;
      cur  $\stackrel{s}{\leftarrow}$  [1, N];
    t[p] ← ct;
    cp[cur] ← 1;
    X ← X + t[p];
  return X

```

**Figure 10.** Coupon collector

ants used by our proof are more intuitive and easier to discover than the equations used in the pGCL proof. Moreover, we note that Ellora naturally supports reasoning about independence, and could be used to reason about the variance of  $X$ , whereas reasoning about independence in pGCL remains possible but challenging.

**Limited randomness.** *pairwise independence* says that if we see the result of  $X_i$ , we do not gain information about all other variables  $X_k$ . However, if we see the result of two variables  $X_i, X_j$ , we may gain information about  $X_k$ .

```

proc pwInd (N : int) :
  var bool x[2N], B[N];
  for i ← 1 to N do
    B[i]  $\stackrel{s}{\leftarrow}$  Ber(1/2);
  for j ← 1 to 2N do
    X[j] ← 0;
    for k ← 1 to N do
      if k ∈ bits(j) then
        X[j] ← X[j] ⊕ B[k]
  return X

```

**Figure 11.** Pairwise Indep.

The proof uses the following fact, which we fully verify: for a uniformly distributed Boolean random variable  $Y$ , and a random variable  $Z$  of any type,

$$Y \# Z \Rightarrow Y \oplus f(Z) \# g(Z) \quad (1)$$

for any two Boolean functions  $f, g$ . Then, note that  $x[i] = \bigoplus_{\{j \in \text{bits}(i)\}} B[j]$  where the big XOR operator ranges over the indices  $j$  where the bit representation of  $i$  has bit  $j$  set. For any two  $i, k \in [1, \dots, 2^N]$  distinct, there is a bit position in  $[1, \dots, N]$  where  $i$  and  $k$  differ; call this position  $r$  and

suppose it is set in  $i$  but not in  $k$ . By rewriting,

$$x[i] = B[r] \oplus \bigoplus_{\{j \in \text{bits}(i) \setminus r\}} B[j] \quad \text{and} \quad x[k] = \bigoplus_{\{j \in \text{bits}(k) \setminus r\}} B[j].$$

Since  $B[j]$  are all independent,  $x[i] \# x[k]$  follows from Eq. (1) taking  $Z$  to be the distribution on tuples  $\langle B[1], \dots, B[N] \rangle$  excluding  $B[r]$ . This verifies pairwise independence:

$$\{\mathcal{L}\} \text{ pwInd}(N) \{ \mathcal{L} \wedge \forall i, k \in [2^N]. i \neq k \Rightarrow x[i] \# x[k] \}.$$

**Adversarial programs.** Pseudorandom functions (PRF) and pseudorandom permutations (PRP) are two idealized primitives that play a central role in the design of symmetric-key systems. Although the most natural assumption to make about a blockcipher is that it behaves as a pseudorandom permutation, most commonly the security of such a system is analyzed by replacing the blockcipher with a perfectly random function. The PRP/PRF Switching Lemma [7, 21] fills the gap: given a bound for the security of a blockcipher as a pseudorandom function, it gives a bound for its security as a pseudorandom permutation.

**Lemma 14** (PRP/PRF switching lemma). *Let  $A$  be an adversary with blackbox access to an oracle  $O$  implementing either a random permutation on  $\{0, 1\}^l$  or a random function from  $\{0, 1\}^l$  to  $\{0, 1\}^l$ . Then the probability that the adversary  $A$  distinguishes between the two oracles in less than  $q$  calls is bounded by  $\frac{q(q-1)}{2^{l+1}}$ :*

$$\left| \Pr_{\text{PRP}}[b \wedge |H| \leq q] - \Pr_{\text{PRF}}[b \wedge |H| \leq q] \right| \leq \frac{q(q-1)}{2^{l+1}}$$

where  $H$  is a map storing each call performed by the adversary and  $|H|$  the size of  $H$ .

Proving this lemma can be done using the Fundamental Lemma of Game-Playing, and bounding the probability of *bad* in the program from Fig. 12. We focus on the latter. Here we apply the [ADV] rule of Ellora with the invariant  $\forall k, \Pr[\text{bad} \wedge |H| \leq k] \leq \frac{k(k-1)}{2^{l+1}}$  where  $|H|$  is the size of the map  $H$ , i.e. the number of adversary call. Intuitively, the invariant says that at each call to the oracle the probability that *bad* has been set before and that the number of adversary call is less than  $k$  is bounded by a polynomial in  $k$ .

The invariant is  $d$ -closed and true before the adversary call, since at that point  $\Pr[\text{bad}] = 0$ . Then we need to prove that the oracle preserves the invariant, which can be done easily using the precondition calculus ([PC] rule).

## 8. Implementation and mechanization

We have built a prototype implementation of Ellora within EasyCrypt [3, 4], a tool-assisted framework originally designed for verifying proofs of cryptographic protocols. EasyCrypt provides a convenient environment for constructing proofs in various Hoare logics, supporting interactive, tactic-based proofs for manipulating assertions and allowing users

```

var H: ( $\{0, 1\}^l$ ,  $\{0, 1\}^l$ ) map;
proc orcl (q:  $\{0, 1\}^l$ ):
  var a:  $\{0, 1\}^l$ ;
  if q  $\notin$  H then
    a  $\xleftarrow{\$}$   $\{0, 1\}^l$ ;
    bad  $\leftarrow$  bad || a  $\in$  codom(H);
    H[q]  $\leftarrow$  a;
  return H[q];

proc main():
  var b: bool;
  bad  $\leftarrow$  false;
  H  $\leftarrow$  [];
  b  $\leftarrow$  A();
  return b;

```

Figure 12. PRP/PRF game

to invoke external tools, like SMT-solvers, to discharge simpler proof obligations. Moreover, EasyCrypt provides a mature set of libraries for both data structures (sets, maps, lists, arrays, etc.) and formalized mathematical theorems (algebra, real analysis, etc.), which we extended with theorems from probability theory. We used the implementation for verifying many examples from the literature, including all the programs presented in § 7 as well as some additional examples (such as polynomial identity test, private running sums, properties about random walks, etc.). The verified proofs bear a strong resemblance to the existing, paper proofs.

## 9. Related work

There is a long tradition of research on verification of probabilistic programs.

*More on Expectation-based techniques.* Expectation-based techniques are one of the most well-studied ideas for deductive verification of probabilistic programs. One of the first such formalisms was *Probabilistic Propositional Dynamical Logic* (PPDL), proposed by Kozen [26], drawing a close analogy to Propositional Dynamical Logic. The central objects in this logic are real-valued functions  $f, g$  on program states, along with rules for constructing and reasoning about the expected value of  $f$  and  $g$  on the output distribution of a program  $c$ . In a series of works initiated by Morgan et al. [31] and described in their textbook [28], McIver, Morgan, and their collaborators extended ideas from PPDL to study an imperative language with probabilistic choice and non-determinism called *probabilistic Guarded Command Language* (pGCL). Like PPDL, pGCL reasons about the expected value of a single real-valued function on program states. The central tool of pGCL is *greatest pre-expectation*, a mechanical procedure similar to Dijkstra’s weakest-pre-condition but transforming expectations instead of predicates. Many subsequent works build on pGCL [15, 16, 20, 23] or use related ideas [1, 22]. In particular, Kaminski et al. [22] give a calculus for bounding expected running time, with support for probabilistic loops that terminate almost surely. They also analyze the coupon collector example of § 7.

*Program logics for probabilistic programs.* Instead of reasoning about a single probability or expected value, a different line of research investigates Hoare logics for probabilistic programs, where the pre-condition and post-condition are prob-

abilistic assertions about the input and output distributions. The earliest system is due to Ramshaw [33], who proposes a program logic where assertions can be formulas involving *frequencies*, essentially probabilities on sub-distributions. Ramshaw’s logic allows assertions to be combined with operators like  $\oplus$ , similar to our approach. More recently, den Hartog [13] presents a Hoare-style logic with supporting general assertions on the distribution, allowing expected values and probabilities. However his **while** rule is based on a semantic condition on the guarded loop body, which is less desirable for verification because it requires reasoning about the semantics of programs. Chadha et al. [8] give decidability results for a probabilistic Hoare logic without **while** loops. We are not aware of any existing system that supports assertions about general expected values; existing works also restrict to Boolean distributions. Rand and Zdancewic [34] formalize a Hoare logic for probabilistic programs but unlike our work, their assertions are interpreted on *distributions* rather than sub-distributions. For conditionals, their semantics rescales the distribution of states that enter each branch. However, their assertion language is restricted and they impose strong restrictions on loops.

*Other approaches.* There have been many other significant works to verify probabilistic program using different formal approaches. For instance, verification of Markov transition systems goes back to at least Hart et al. [17], Sharir et al. [36]; our condition for ensuring almost-sure termination in loops is directly inspired by their work. Automated methods include model checking (see e.g., [2, 24, 27]) and abstract interpretation (see e.g., [12, 29]). For analyzing probabilistic loops in particular, there are tools for reasoning about running time. There are also automated systems for synthesizing invariants [5, 11]. Chakarov and Sankaranarayanan [9, 10] use a martingale method to compute the expected time of the coupon collector process for  $N = 5$ —fixing  $N$  lets them focus on a program where the outer **while** loop is fully unrolled. Martingales are also used by Fioriti and Hermanns [14] for analyzing probabilistic termination. Finally, there are approaches involving symbolic execution; Sampson et al. [35] use a mix of static and dynamic analysis to check probabilistic programs from the approximate computing literature.

## 10. Conclusion and perspective

Ellora is a general-purpose framework for verification of randomized programs. We have proved its soundness, and its expressiveness through representative examples from the literature. Prime targets for future formalization include accuracy of differentially private algorithms, lower bounds, distributed algorithms, and amortized complexity. We also hope to apply Ellora to more mathematical areas, like combinatorics proofs based on the probabilistic method. Finally, we plan to extend and automate the IL logic.

## References

- [1] P. Audebaud and C. Paulin-Mohring. Proofs of randomized algorithms in Coq. *Sci. Comput. Program.*, 74(8):568–589, 2009.
- [2] C. Baier. Probabilistic model checking. In J. Esparza, O. Grumberg, and S. Sickert, editors, *Dependable Software Systems Engineering*, volume 45 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 1–23. IOS Press, 2016.
- [3] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin. Computer-aided security proofs for the working cryptographer. In P. Rogaway, editor, *Advances in Cryptology - CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, 2011.
- [4] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P. Strub. Easycrypt: A tutorial. In A. Aldini, J. Lopez, and F. Martinelli, editors, *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2013.
- [5] G. Barthe, T. Espitau, L. M. Ferrer Fioriti, and J. Hsu. Synthesizing probabilistic invariants via Doob’s decomposition. In *International Conference on Computer Aided Verification (CAV), Toronto, Ontario*, 2016. To appear.
- [6] G. Barthe, M. Gaboardi, B. Grégoire, J. Hsu, and P.-Y. Strub. A program logic for union bounds. In *International Colloquium on Automata, Languages and Programming (ICALP), Rome, Italy*, 2016. To appear.
- [7] M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In S. Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426. Springer, 2006. ISBN 3-540-34546-9.
- [8] R. Chadha, L. Cruz-Filipe, P. Mateus, and A. Sernadas. Reasoning about probabilistic sequential programs. *Theoretical Computer Science*, 379(1-2):142–165, 2007.
- [9] A. Chakarov and S. Sankaranarayanan. Probabilistic program analysis with martingales. In *International Conference on Computer Aided Verification (CAV), Saint Petersburg, Russia*, volume 8044 of *Lecture Notes in Computer Science*, pages 511–526. Springer, 2013.
- [10] A. Chakarov and S. Sankaranarayanan. Expectation invariants as fixed points of probabilistic programs. In *Static Analysis Symposium (SAS)*, volume 8723 of *Lecture Notes in Computer Science*, pages 85–100. Springer-Verlag, 2014.
- [11] K. Chatterjee, H. Fu, P. Novotný, and R. Hasheminezhad. Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), St Petersburg, Florida*, pages 327–342, 2016.
- [12] P. Cousot and M. Monerau. Probabilistic abstract interpretation. In H. Seidl, editor, *21st European Symposium on Programming, ESOP 2012*, volume 7211 of *Lecture Notes in Computer Science*, pages 169–193. Springer, 2012.
- [13] J. den Hartog. *Probabilistic extensions of semantical models*. PhD thesis, Vrije Universiteit Amsterdam, 2002.
- [14] L. M. F. Fioriti and H. Hermans. Probabilistic termination: Soundness, completeness, and compositionality. In S. K. Rajamani and D. Walker, editors, *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages, POPL 2015*, pages 489–501. ACM, 2015.
- [15] F. Gretz, J. Katoen, and A. McIver. Prinsys - on a quest for probabilistic loop invariants. In *Quantitative Evaluation of Systems - 10th International Conference, QEST 2013*, pages 193–208, 2013.
- [16] F. Gretz, N. Jansen, B. L. Kaminski, J. Katoen, A. McIver, and F. Olmedo. Conditioning in probabilistic programming. In *Mathematical Foundations of Programming Semantics*, 2015.
- [17] S. Hart, M. Sharir, and A. Pnueli. Termination of probabilistic concurrent programs. *ACM Trans. Program. Lang. Syst.*, 5(3): 356–380, 1983.
- [18] J. Hurd. Formal verification of probabilistic algorithms. Technical Report UCAM-CL-TR-566, University of Cambridge, Computer Laboratory, 2003.
- [19] J. Hurd. Verification of the miller-rabin probabilistic primality test. *J. Log. Algebr. Program.*, 56(1-2):3–21, 2003.
- [20] J. Hurd, A. McIver, and C. Morgan. Probabilistic guarded commands mechanized in HOL. *Theor. Comput. Sci.*, 346(1): 96–112, 2005.
- [21] R. Impagliazzo and S. Rudich. Limits on the provable consequences of one-way permutations. In D. S. Johnson, editor, *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washington, USA*, pages 44–61. ACM, 1989. ISBN 0-89791-307-8.
- [22] 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.
- [23] 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.
- [24] J.-P. Katoen. The probabilistic model-checking landscape. In *Proceedings of LICS’16*, 2016.
- [25] D. Kozen. Semantics of probabilistic programs. In *20th IEEE Symposium on Foundations of Computer Science, FOCS 1979*, pages 101–114. IEEE Computer Society, 1979.
- [26] D. Kozen. A probabilistic PDL. *J. Comput. Syst. Sci.*, 30(2): 162–178, 1985.
- [27] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV’11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.
- [28] A. McIver and C. Morgan. *Abstraction, Refinement, and Proof for Probabilistic Systems*. Monographs in Computer Science.

Springer, 2005.

- [29] D. Monniaux. Abstract interpretation of probabilistic semantics. In J. Palsberg, editor, *Static Analysis, 7th International Symposium, SAS 2000*, volume 1824 of *Lecture Notes in Computer Science*, pages 322–339. Springer, 2000.
- [30] C. Morgan. Proof rules for probabilistic loops. In *Proceedings of the BCS-FACS 7th Conference on Refinement, FAC-RW'96*, 1996.
- [31] C. Morgan, A. McIver, and K. Seidel. Probabilistic predicate transformers. *ACM Trans. Program. Lang. Syst.*, 18(3):325–353, 1996.
- [32] J. Pearl and A. Paz. Graphoids: Graph-based logic for reasoning about relevance relations. In *ECAI*, pages 357–363, 1986.
- [33] L. H. Ramshaw. *Formalizing the Analysis of Algorithms*. PhD thesis, Computer Science, 1979.
- [34] R. Rand and S. Zdancewic. VPHL: A Verified Partial-Correctness Logic for Probabilistic Programs. In *Mathematical Foundations of Program Semantics (MFPS XXXI)*, 2015.
- [35] A. Sampson, P. Panckheha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze. Expressing and verifying probabilistic assertions. In M. F. P. O’Boyle and K. Pingali, editors, *ACM Conference on Programming Language Design and Implementation, PLDI '14*, page 14. ACM, 2014.
- [36] M. Sharir, A. Pnueli, and S. Hart. Verification of probabilistic programs. *SIAM J. Comput.*, 13(2):292–314, 1984.
- [37] L. G. Valiant. A scheme for fast parallel communication. *SIAM journal on computing*, 11(2):350–361, 1982.
- [38] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing, STOC '81*, pages 263–277, New York, NY, USA, 1981. ACM.
- [39] P. R. M. Vásquez, N. Wolovick, and P. R. D’Argenio. Probabilistic hoare-like logics in comparison. Technical report, Universidad Nacional de Córdoba, 2004.