A Program Logic for Concurrent Randomized Programs in the Oblivious Adversary Model

Weijie Fan[1], Hongjin Liang[1]✉, Xinyu Feng[1], and Hanru Jiang[2]

[1] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, Jiangsu, China
weijiefan@smail.nju.edu.cn, {hongjin, xyfeng}@nju.edu.cn
[2] Beijing Institute of Mathematical Sciences and Applications, Beijing 101408, Beijing, China
hanru@bimsa.cn

Abstract. Concurrent randomized programs in the oblivious adversary model are extremely difficult for modular verification because the interaction between threads is very sensitive to the program structure and the execution steps. We propose a new program logic supporting thread-local verification. With a novel "split" mechanism, one can split the state distribution into smaller partitions, and the reasoning can be done based on each partition independently, which allows us to avoid considering different execution paths of branch statements simultaneously. The logic rules are compositional and are natural extensions of their sequential counterparts. Using our program logic, we verify four typical algorithms in the oblivious adversary model.

1 Introduction

Randomization has become an important and powerful technique in the design of concurrent and distributed algorithms. By introducing probabilistic coin-flip operations, problems like consensus and leader election can be solved efficiently (e.g. [12,2,3]), despite being inherently difficult or even impossible to solve in a non-probabilistic concurrent setting.

To understand the semantics of concurrent randomized programs, one has to take into account the interplay between concurrency and randomization. In particular, one must answer the question: can the result of a coin-flip operation affect the choice of scheduling (i.e. which thread will perform the next operation)? For this, algorithm designers propose a spectrum of *adversary models* specifying the knowledge about the past execution that a scheduler (a.k.a. an adversary) can use for choosing the next thread. Different adversary models differs in the amount of knowledge they assume, varying from none to all.

At one end of the spectrum is the *oblivious adversary* (OA) model, where an adversary has no knowledge and must fix the entire schedule prior to the execution. The OA model is a natural abstraction of most real-world scheduling

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-3-031-91118-7_13

V. Vafeiadis (Ed.): ESOP 2025, LNCS 15694, pp. 322–348, 2025.
https://doi.org/10.1007/978-3-031-91118-7_13

algorithms, including the round-robin scheduling and the priority-based scheduling. It reflects the scheduling in almost all real general infrastructures such as operating systems or programming languages (e.g. as in golang) where the scheduling does not rely on the specific behaviors of the threads being scheduled.

Designing algorithms for the OA model has gained lots of attention and more than ten algorithms have been proposed over the years (see [4,5] for a comprehensive introduction). As a concrete example, consider Chor et al. [12]'s *conciliator* algorithm. A conciliator is a weak consensus object that guarantees probabilistic agreement, namely that with a high probability the return values of all threads are equal. In Chor et al. [12]'s conciliator algorithm, each thread i executes C_i:

$$C_i \stackrel{\text{def}}{=} (\textbf{while } (s = 0) \textbf{ do } \langle s := i \rangle \oplus_p \langle \textbf{skip} \rangle)\,; y_i := s$$

Here s is a shared variable initialized to 0, y_i is the local variable for thread i that records its return value. The probabilistic choice $\langle s := i \rangle \oplus_p \langle \textbf{skip} \rangle$ says that thread i writes i to s with probability p and does nothing (**skip**) with probability $1 - p$. It repeats until the thread observes $s \neq 0$, then it loads s to y_i. Given n threads running the conciliator code in the OA model, the algorithm ensures the postcondition $\mathbf{Pr}(y_1 = y_2 = \cdots = y_n) \geq (1-p)^{n-1}$, i.e., the probability for the threads to reach a consensus (thus $y_1 = y_2 = \cdots = y_n$) is no less than $(1-p)^{n-1}$.

However, there has been little attention paid to verifying algorithms in the OA model. Existing program logics for verifying concurrent randomized programs [20,18,14] work with only the *strong adversary* (SA) model, which is at the other end of the spectrum of adversary models. A strong adversary has the full knowledge of the past execution, including outcomes of past coin-flips, thread-local states and shared states. Consequently, any algorithm which is correct under SA must still be correct under OA, but not vice versa. For instance, the aforementioned conciliator algorithm is *not* correct in SA and we will explain why in Sec. 6. None of the existing program logics can apply to the conciliator, or more generally, to any algorithms which are correct only with weaker adversaries such as OA.

On the one hand, it is unclear how to *take advantage of* the OA model in the verification. On the other hand, the OA model brings its own verification challenges. As we will see in Sec. 3, the program behaviors in the OA model seem sensitive to the number of execution steps in different program branches, but the verification with program logics should be modular, syntax-directed and insensitive to the number of execution steps.

The good news is, from the existing algorithms designed for the OA model, we observe that the correctness properties of these algorithms usually follow certain common patterns and can be specified by what we call "closed" assertions, which will be introduced in Sec. 3.2. To verify these properties, we do not need to prove they hold over the whole state distribution, which may contain states resulting from the execution of different program branches. Instead, we can prove there exists a partition of the distribution such that the property holds over every part. For *closed* assertions, the validity over every part implies the validity over the whole distribution.

Based on this observation, we propose the first program logic for concurrent randomized programs targeting the OA model. Our work makes the following new contributions:

- We take advantage of the OA model by proposing an abstract small-step operational semantics over state distributions, which allows us to apply classical concurrency reasoning techniques (such as invariants) by interpreting assertions over state distributions.
- We propose a novel proof technique called *split* to support modular reasoning and overcome the problem with branch statements. By splitting a state distribution into several smaller ones, we can reason about the different program branches independently. This leaves us only to prove the postcondition holds over a partition of the final state distribution. Then we can derive it for the whole distribution as long as the postcondition is closed.
- We design a set of logic rules for compositional reasoning about concurrent randomized programs with the split mechanism. Thanks to the split idea, our rules for sequential composition, **if**-statements and **while**-loops are simple and natural extensions of their classical (non-probabilistic) counterparts.
- We prove that our logic ensures partial correctness of concurrent randomized programs where the adversaries are also randomized. Since we focus on closed assertions as postconditions, the verification is independent of the distribution of schedules. The partial correctness verified by the logic holds over arbitrary probabilistic distributions of oblivious adversaries.
- Using our logic, we report the first formal verification of four typical algorithms in the OA model, including the aforementioned conciliator [12], group election (the core phase of Alistarh and Aspnes' randomized test-and-set algorithm [2]), a shared three-sided dice and a multiplayer level-up game.

Outline. Below we first review mathematical preliminaries in Sec. 2. Then we informally explain our key ideas in Sec. 3. We present the language setting including our abstract semantics in Sec. 4. We develop our program logic in Sec. 5, and verify conciliator as a case study in Sec. 6. We discuss related work in Sec. 7. The accompanying technical report (TR) [13] contains the full formal details, including semantics rules, logic rules and soundness proofs, and examples.

2 Preliminaries

Below we review the background on probability theory and sketch the basic mathematical notations used in our work for describing probabilities, expected values, etc. Readers who are not interested in mathematics can safely skip this section and come back later when the notations are used.

A *sub-distribution* over a set A is defined as a function $\mu : A \to [0, 1]$ such that

- the support $supp(\mu) \stackrel{\text{def}}{=} \{a \in A \mid \mu(a) > 0\}$ is countable; and
- the weight $|\mu| \stackrel{\text{def}}{=} \sum_{a \in A} \mu(a)$ is less than or equal to 1.

If we have $|\mu| = 1$, we say μ is a *distribution* over A. We use $\mathbb{SD}_A$ to denote the set of sub-distributions over A, and $\mathbb{D}_A$ to denote the set of distributions. For $\mu \in \mathbb{SD}_A$, intuitively $\mu(a)$ represents the *probability* of drawing a from μ.

We define the *probability of an event* $E : A \to \text{Prop}$ and the *expected value of a random variable* $V : A \to \mathbb{R}$ as follows, denoted by $\mathbf{Pr}_{a\sim\mu}[E(a)]$ and $\mathbb{E}_{a\sim\mu}[V(a)]$ respectively (where a is a bound variable, just like $\sum_{a\in A} f(a)$). Here Prop represents the set of propositions, and $\mathbb{R}$ is the set of real numbers.

$$\mathbf{Pr}_{a\sim\mu}[E(a)] \stackrel{\text{def}}{=} \sum_{a\in A}\{\mu(a) \mid E(a)\} \qquad \mathbb{E}_{a\sim\mu}[V(a)] \stackrel{\text{def}}{=} \sum_{a\in A} \mu(a)\cdot V(a) \tag{1}$$

For instance, suppose μ is a state distribution, and $\mathbf{q}$ is a state assertion (we write $\sigma \models \mathbf{q}$ if $\mathbf{q}$ holds at the state σ). Then $\mathbf{Pr}_{\sigma\sim\mu}[\sigma \models \mathbf{q}]$ represents the probability that $\mathbf{q}$ is satisfied. If $[\![e]\!]_\sigma$ is the evaluation of the expression e on σ, then $\mathbb{E}_{\sigma\sim\mu}[[\![e]\!]_\sigma]$ represents the expected value of e in μ.

For an event E with non-zero probability in μ (i.e. $\mathbf{Pr}_{a\sim\mu}[E(a)] > 0$), we define the *conditional sub-distribution* $\mu|_E$ as follows:

$$\mu|_E \stackrel{\text{def}}{=} \lambda a. \begin{cases} \frac{\mu(a)}{\mathbf{Pr}_{a\sim\mu}[E(a)]}, & \text{if } E(a) \text{ holds} \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

Given two sub-distributions $\mu_1, \mu_2 \in \mathbb{SD}_A$ and a probability $p \in [0, 1]$, we define the *mixture sub-distribution* $\mu_1 \oplus_p \mu_2 \in \mathbb{SD}_A$ as follows:

$$\mu_1 \oplus_p \mu_2 \stackrel{\text{def}}{=} \lambda a.\ p\cdot\mu_1(a) + (1-p)\cdot\mu_2(a) \tag{3}$$

Given two sub-distributions $\mu_1 \in \mathbb{SD}_A$ and $\mu_2 \in \mathbb{SD}_B$, we define the *product sub-distribution* $\mu_1 \otimes \mu_2 \in \mathbb{SD}_{A\times B}$ as follows:

$$\mu_1 \otimes \mu_2 \stackrel{\text{def}}{=} \lambda(a, b).\ \mu_1(a)\cdot\mu_2(b) \tag{4}$$

In Sec. 4.2, we will use the product $\otimes$ to compute the initial distribution of program configurations, from the initial program $\mathbb{C}$ and an initial state distribution. When $\mathbb{C}$'s execution ends, we will extract the final state distribution from the final distribution of program configurations by projection. Specifically, given $\mu \in \mathbb{SD}_{A\times B}$, the *projection* of μ with the sets A and B is defined as:

$$\mu^{(A)} \stackrel{\text{def}}{=} \lambda a'.\, \mathbf{Pr}_{(a,b)\sim\mu}[a = a'] \qquad \mu^{(B)} \stackrel{\text{def}}{=} \lambda b'.\, \mathbf{Pr}_{(a,b)\sim\mu}[b = b'] \tag{5}$$

For *almost surely terminating* programs (i.e. programs which have infinite executions with zero probability and terminate with probability 1), we define the "final" state distribution as the limit of an infinite sequence of state distributions. In general, we define the limit of a convergent sequence of sub-distributions in Def. 6.

Definition 6 (convergent sequence of sub-distributions). Let A be a set, $\vec{\mu}$ be an infinite sequence of sub-distributions over A. We say $\vec{\mu}$ *converges to a sub-distribution* μ, represented as $\lim \vec{\mu} = \mu$, if and only if $\lim_{n\to\infty} \sum_{a\in A} |\vec{\mu}[n](a) - \mu(a)| = 0$ (where $\vec{\mu}[n]$ means the n-th element of the sequence $\vec{\mu}$). We say $\vec{\mu}$ diverges and $\lim \vec{\mu}$ is undefined if $\vec{\mu}$ does not converge to any μ.

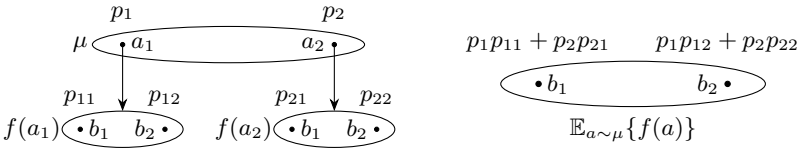


Fig. 1: Expected sub-distribution

Definition 7 (expected sub-distribution). Let $\mu \in \mathbb{SD}_A$ and $f : A \to \mathbb{SD}_B$. The *expected sub-distribution* $\mathbb{E}_{a\sim\mu}\{f(a)\} \in \mathbb{SD}_B$ is defined as

$$\mathbb{E}_{a\sim\mu}\{f(a)\} \stackrel{\text{def}}{=} \lambda b.\, \textstyle\sum_{a\in A} \mu(a) \cdot f(a)(b)$$

Definition 7 computes the sub-distributions' expectation. As illustrated in Fig. 1, the function f transforms each element a_i in the support of μ to a sub-distribution $f(a_i)$, and then the expected sub-distribution (see the right side of the figure) is the mixture of all $f(a_i)$.

Also, from a sub-distributions' sub-distribution $\mu \in \mathbb{SD}_{\mathbb{SD}_A}$, we can compute the *flattened sub-distribution* $\overline{\mu} \in \mathbb{SD}_A$ as the mixture of all the sub-distributions in the support of μ:

$$\overline{\mu} \stackrel{\text{def}}{=} \lambda a.\, \textstyle\sum_{\nu\in supp(\mu)} \mu(\nu) \cdot \nu(a)\,. \tag{8}$$

3 Informal Development

Below we start with reasoning about sequential randomized programs (Sec. 3.1). For concurrent randomized programs, we introduce the oblivious adversary (OA) model and define the correctness of programs with randomized schedules (Sec. 3.2). Then we show how to do thread-local reasoning by taking advantage of OA (Sec. 3.3). To address the challenges posed by branch statements (Sec. 3.4), we propose the split mechanism (Sec. 3.5).

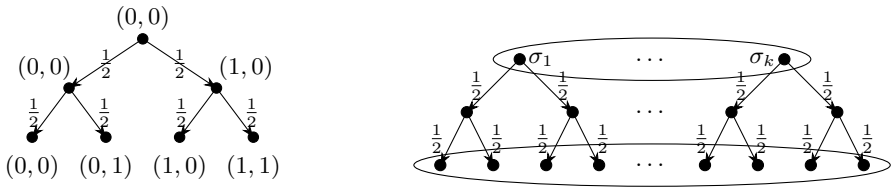
3.1 Sequential Randomized Programs and Their Correctness

Randomized programs can be viewed as programs in a classical (non-probabilistic) programming language (e.g. WHILE) extended with probabilistic choice statements $\langle C_1\rangle \oplus_p \langle C_2\rangle$. It makes a random choice to execute $\langle C_1\rangle$ or $\langle C_2\rangle$, with probability p and $1-p$, respectively. Here we use $\langle C\rangle$ to represent an *atomic* statement that executes C in one step (see the formal semantics in Sec. 4.1).

The execution of a *sequential* randomized program starting from a particular initial state forms a tree. For instance, Fig. 2a shows the execution tree for

$$\mathit{Coins} \stackrel{\text{def}}{=} \langle x := 0\rangle \oplus_{\frac{1}{2}} \langle x := 1\rangle;\ \langle y := 0\rangle \oplus_{\frac{1}{2}} \langle y := 1\rangle;$$

starting from the initial state where x and y are both 0. Each branching in the tree corresponds to a probabilistic choice. If we consider all possible initial states, the execution becomes a forest (where each node represents a program state σ), as shown in Fig. 2b.



(a) (b)

Fig. 2: Execution of a sequential program. In (a), a pair at a node specifies x and y's values in the state.

Correctness. Although the execution model based on the view of state transitions is similar to the model of classical sequential programs, the properties of randomized programs can be significantly different. For the program *Coins*, one may want to derive properties like "the probability that x equals y at the end of the program is 0.5". Unlike a postcondition in Hoare-style logics for classical sequential programs, which is expected to hold over *every* leaf node of the forest, the above property describes *the collection of all the leaf nodes* as a whole, i.e. the state distribution at the end of the program.

Therefore, in the Hoare-style specification $\{P\}C\{Q\}$ for randomized programs, P and Q are assertions over distributions of initial states and final states, respectively. For the example *Coins*, we can specify the aforementioned property as $\{\mathbf{true}\}\mathit{Coins}\{\mathbf{Pr}(x = y) = 0.5\}$ or $\{\mathbf{true}\}\mathit{Coins}\{\lceil x = y\rceil \oplus_{0.5} \lceil x \neq y\rceil\}$. Here $\lceil\mathbf{p}\rceil$ lifts the state assertion $\mathbf{p}$ to an assertion over *state distributions* μ, requiring that $\mathbf{p}$ holds at all states in $\mathit{supp}(\mu)$. The assertion $P \oplus_p Q$ holds at μ, if μ is a *mixture* of two distributions μ_0 and μ_1, which are associated with probabilities p and $1{-}p$, and satisfy P and Q respectively. We can give the following Hoare-logic rule to probabilistic choices:

$$\frac{\vdash_{\text{SQ}} \{P\}C_1\{Q_1\} \qquad \vdash_{\text{SQ}} \{P\}C_2\{Q_2\}}{\vdash_{\text{SQ}} \{P\}\langle C_1\rangle \oplus_p \langle C_2\rangle\{Q_1 \oplus_p Q_2\}} \ (\text{SQ-PCH})$$

In this view, a program C transforms a state distribution μ satisfying P to another state distribution μ' satisfying Q (an alternative view is expectation-based, where P and Q are expectations [17,8]). The resulting logic rules (e.g. [6]) are almost the same as the classical (non-probabilistic) ones — we just need to lift the assertions from predicates over states to predicates over state distributions.

3.2 Concurrent Randomized Programs and the OA Model

A concurrent randomized program $C_1 \parallel \cdots \parallel C_n$ (denoted by $\mathbb{C}$) has two sources of nondeterminism: the probabilistic choices (in each thread C_i) and the scheduling. Its correctness usually assumes a certain class of scheduling, specified by an *adversary model*.

The *oblivious* adversary (OA) model considered in this paper requires that the scheduling must be determined prior to the execution, regardless of the outcomes of a thread's local coin-flip operations. For example, Fig. 3 shows all

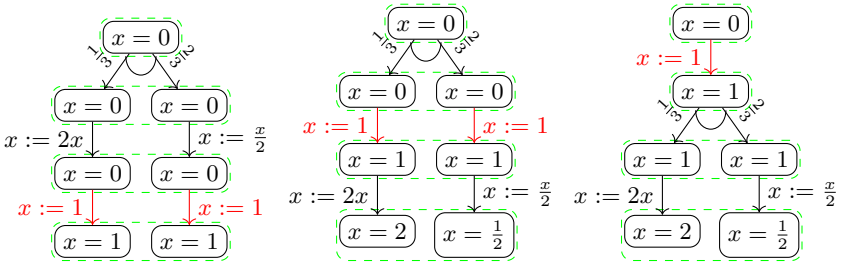


Fig. 3: Execution trees in OA model for $\mathbb{C}_x \stackrel{\text{def}}{=} (\langle x := 2x \rangle \oplus_{\frac{1}{3}} \langle x := \frac{x}{2} \rangle \parallel x := 1)$

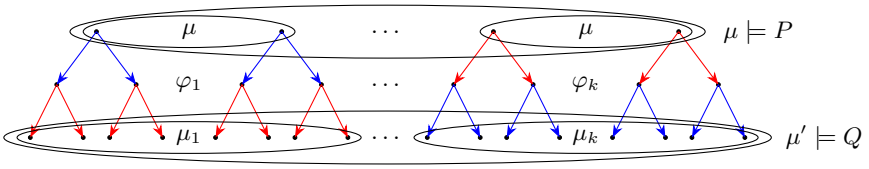


Fig. 4: Illustration of $\models \{P\}\mathbb{C}\{Q\}$

the possible executions in the OA model for a simple program $\mathbb{C}_x$ consisting of two threads: $\langle x := 2x \rangle \oplus_{\frac{1}{3}} \langle x := \frac{x}{2} \rangle \parallel x := 1$. In the concurrent setting, the probabilistic choice $\langle C_1 \rangle \oplus_p \langle C_2 \rangle$ is executed in *two steps*: it first flips a coin, getting heads with probability p and tails with probability $1 - p$, and then executes either the atomic statement $\langle C_1 \rangle$ for heads, or $\langle C_2 \rangle$ for tails.

Therefore, in OA, there are only three possible schedules for $\mathbb{C}_x$: $t_1\ t_1\ t_2$ (Fig. 3a); $t_1\ t_2\ t_1$ (Fig. 3b); and $t_2\ t_1\ t_1$ (Fig. 3c). In the figure, state transitions by different threads are in different colors (in black for t_1, and in red for t_2). We can see that, by fixing a specific OA schedule, the transitions at the same layer of an execution tree must be made by the *same* thread.

In contrast, the *strong* adversary (SA) model allows arbitrary scheduling. An SA scheduler has the full knowledge of machine states, especially including the outcomes of coin-flip operations, and can rely on that knowledge to schedule threads. For the example $\mathbb{C}_x$, in addition to the three schedules in Fig. 3, the SA model also allows two additional schedules, where t_1 and t_2 are scheduled in different orders for different outcomes of the coin flip. As such, the transitions at the same layer of an execution tree could be made by *different* threads.

This example also demonstrates that, thanks to the restriction of the scheduling, one can derive stronger properties of programs in the OA model that do not hold in the SA model. As shown in Fig. 3, in the OA model, the expected value of x at the end of execution is 1, which is not true considering the two more schedules in the SA model.

Correctness and closed assertions. What is the meaning of the Hoare triple $\{P\}\mathbb{C}\{Q\}$ now? Figure 4 shows the execution of a concurrent program, where μ

is the distribution of the initial states. We use $\mu \models P$ to denote that μ satisfies P, which will be formally defined in Sec. 5.1. The execution under each (OA) schedule φ_i corresponds to a forest, as in the case for sequential programs. Edges of different colors represent execution steps from different threads. The execution under all schedules forms *a set of forests*. It is obvious that P specifies μ, but what about Q?

Here we have two choices. We can either view the schedules as being *non-deterministic*, or as being *probabilistic*. For the former, we require that Q holds over every μ_i (the leaf node distribution of the forest generated with the schedule φ_i). However, this result is not strong enough — if we sample the execution of $\mathbb{C}$ and observe the final results, the sampled executions may not be generated with the same schedule, that is, the final states we observe may come from different μ_i. So it is more natural to take the latter (probabilistic) view of schedule and consider the mixture distribution μ' of $\mu_1, \ldots, \mu_k, \ldots$, where the weight of each μ_i is the probability of the schedule φ_i. Since we do not know the distribution of schedules in advance, Q needs to hold for all schedule distributions, that is, Q holds over μ' obtained by taking an *arbitrary* probability distribution for $\mu_1, \ldots, \mu_k, \ldots$

We use $\models_{\text{ND}} \{P\}\mathbb{C}\{Q\}$ to represent the semantics of the Hoare triple under the *non-deterministic* view, and $\models_{\text{PR}} \{P\}\mathbb{C}\{Q\}$ for the *probabilistic* view. It is easy to prove the latter implies the former. The reverse does not hold in general, but it holds if Q is "closed". Here $\mathbf{closed}(Q)$ requires that the mixture of any (potentially countably infinite) number of distributions satisfies Q if each of these distributions satisfies Q. (We will formally define $\mathbf{closed}(Q)$ in Sec. 5.1.) As a result, for a closed postcondition, we can reduce the proof of $\models_{\text{PR}} \{P\}\mathbb{C}\{Q\}$ to the proof of $\models_{\text{ND}} \{P\}\mathbb{C}\{Q\}$.

As far as we know, most concurrent randomized algorithms have closed post-conditions. As examples of closed assertions, $\lceil b \rceil$, $\mathbf{Pr}(b) = 0.5$ and $\mathbb{E}(x) = 1 \wedge \lceil x \geq 0 \rceil$ are all closed. So, for the earlier example $\mathbb{C}_x$, it suffices to prove that the leaf distribution of each execution tree in Fig. 3 satisfies $\mathbb{E}(x) = 1 \wedge \lceil x \geq 0 \rceil$.

We give the formal definition of $\models_{\text{ND}} \{P\}\mathbb{C}\{Q\}$ in Sec. 4.1. We show the formal definition of $\models_{\text{PR}} \{P\}\mathbb{C}\{Q\}$ and prove that they are equivalent when Q is closed in the TR [13]. In this paper we focus on closed Q's only and omit the subscript ND/PR henceforth. Note that $\mathbf{closed}(Q)$ is *not* an overly strong requirement for practical programs, because it is needed only for the postcondition Q of the *whole program* $\mathbb{C}$. The postconditions for individual statements and threads do not need to be closed.

3.3 Thread-Local Reasoning in OA

The question is, *how to take advantage of the OA model and verify the stronger correctness guarantee of a program by thread-local reasoning, i.e., verifying one thread at a time.*

A natural thought is to extend the sequential reasoning in Sec. 3.1 to concurrency. To this end, we hope to view the execution of a concurrent program as transitions over state distributions, as we do for sequential reasoning. However,

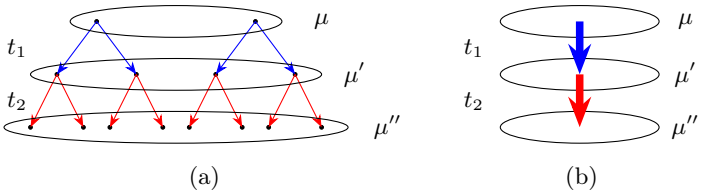


Fig. 5: Concrete *vs.* Abstract Operational Semantics in OA

unlike sequential semantics that are usually *big-step* (see e.g. [6,19]) and care about only the initial and final state distributions, the transitions in a concurrent setting need to be small-step, to reflect the interleaving between threads.

One might also consider to migrate the existing approaches for the SA model to the OA setting. However, the interleaving pattern between threads in the OA model is very different from that in the SA model. The SA model allows that different threads may be scheduled for different outcomes of a probabilistic choice operation, while the OA model does not allow it. As a result, program logics for SA (e.g. [18,14]) adopt *weak* assumptions on the environment behaviors in the thread view: for different states in the support of the current state distribution, different environment threads may interrupt and take very different steps. Therefore, they model the environment behaviors as transitions from states to state distributions (e.g. [18]) or transitions from states to states (e.g. [14]).

However, this idea may not be as useful in the OA setting as in the SA setting (thought it is still sound). Algorithms in the OA model usually rely on the assumption that the scheduling cannot depend on the results of probabilistic choices, so the weak assumption that different states may be interrupted by different environment threads is too weak in the OA setting, and it is not obvious how to forbid the impossible interleavings in the OA model if we still model the environment behaviors as transitions from states to state distributions or transitions from states to states.

To address this problem, we exploit the stronger assumption on the environment behaviors: for different states in the support of the current distribution, it must be the same environment thread that interrupts and take steps. Therefore, we propose an abstract operational semantics and layer-based reasoning.

Abstract operational semantics. In the OA model, we observe that, for all the states at the same layer of the execution forest (i.e. nodes of the same depths, as shown in Fig. 5a), it is always the same thread picked to execute the next step, since the schedule is predetermined. That is, the edges with the same depth are always of the same color, representing steps from the same thread. Naturally, we can view the states of the same layer as a whole, forming a state distribution. If we also view the edges between two layers as a whole, then Fig. 5a is abstracted to Fig. 5b. This gives us an *abstract operational semantics* with small-step transitions over state distributions. The execution looks like an interleaving execution of a classical (non-probabilistic) concurrent program.

Consequently, we can apply classical concurrency reasoning techniques (e.g. invariants) to reason about executions in our abstract semantics. Our abstraction is sound in that the Hoare-triple $\{P\}\mathbb{C}\{Q\}$ valid in our abstract semantics also holds with the concrete semantics.

Invariants. To do thread-local reasoning, one needs to specify the interference between the current thread and its environment (i.e. the other threads), which can be modeled by an invariant I. For classical concurrent programs, I is a state assertion that needs to hold at all times. The current thread can assume that I holds before each of its steps, but it must also ensure that I still holds after each step. For a randomized program, we define I over state distributions. It holds at all the μ's in executions in our abstract semantics (e.g. μ, μ' and μ'' in Fig. 5b). Since every such μ corresponds to a layer in the concrete semantics, we call I a *layer invariant* and the reasoning layer-based.

In addition to layer invariants I, our logic also uses *non-probabilistic rely-guarantee conditions* (R and G), to simplify the formulation of I in proofs of programs. By "non-probabilistic", we mean that R and G specify state transitions in the concrete semantics (but do not specify the probability of the transitions). Their treatment is the same as in classical rely-guarantee reasoning [15].

Unfortunately, we need to address one more challenge to make this nice abstraction work. To define the abstract operational semantics, we view all the edges (program steps) at the same layer in Fig. 5a as a whole to get Fig. 5b. However, although these edges are from the same thread, they may still correspond to the execution of *different code*, due to the branch statements in the thread. Below we explain the challenges and our solution in detail.

3.4 Problems with Branch Statements

A program may contain branch statements such as **if**-statements and **while**-loops, which condition on random variables (i.e. variables whose values are probabilistic). Different branches may take different numbers of steps to execute, making it difficult to do layer-based reasoning.

For instance, we consider the program $C \parallel c_4$, where:

$$C \stackrel{\text{def}}{=} (\textbf{if } (x = 0) \textbf{ then } (c_{11}; c_{12}) \textbf{ else } c_{21}); c_3;$$

Here each $c_\square$ stands for an atomic command. Assume the initial values of x are assigned in a probabilistic choice, which is either 0 or 1. Figure 6 shows a possible execution, where we need to consider the two possibilities corresponding to the two initial values of x. Note that we allow the right branch to execute **skip** when it reaches the end while the left branch executes c_3.

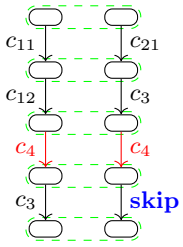


Fig. 6

Thread t_1 switches to t_2 after executing two steps (we omit the step evaluating the boolean condition). The layer-based reasoning asks us to find some invariant and prove that it holds over the distribution of every layer (i.e. every

green dashed box). This forces us to consider the simultaneous execution of c_{11} in the **then**-branch and c_{21} in the **else**-branch. Even worse, since the two branches have different lengths, we have to consider the simultaneous execution of c_{12} and c_3. This looks particularly unreasonable if we consider the fact that c_3 actually sequentially follows c_{12} in the program structure! This makes it almost impossible to design structural and compositional Hoare-style logic rules. The problem is exacerbated by **while**-loops, where the number of rounds of loops may rely on random variables.

Note that this problem does not show up in the deterministic setting where there is no randomization and we prove properties of individual states. In the execution of **if**-statements, a state either enters the **then**-branch or enters the **else**-branch, but not both. So we only need to verify the two cases respectively.

We also do not have to worry about the problem with branch statements in the sequential probabilistic setting. Since there is no interleaving, we can reason about probabilistic properties in a "big-step" flavor where we only consider the initial state distribution and the final one. To reason about the branch statement, we can reason about the different branches (on the corresponding sub-distributions) separately and then do a mixture at the join point, as shown by the (COND) rule in Barthe et al. [6]'s sequential logic:

$$\frac{\{P_1 \wedge \lceil b \rceil\} C_1 \{Q_1\} \qquad \{P_2 \wedge \lceil \neg b \rceil\} C_2 \{Q_2\}}{\{(P_1 \wedge \lceil b \rceil) \oplus (P_2 \wedge \lceil \neg b \rceil)\} \textbf{if } (b) \textbf{ then } C_1 \textbf{ else } C_2 \{Q_1 \oplus Q_2\}} \ (\text{COND})$$

The (COND) rule in [6] is sound for sequential programs, but not for the concurrent OA setting. If C_1 and C_2 have different numbers of steps, then $Q_1 \oplus Q_2$ specifies a state distribution where states are not at the same "layer", which will make it difficult to reason about subsequent statements.

Below we use an interesting example to further demonstrate the problem and then introduce our solution.

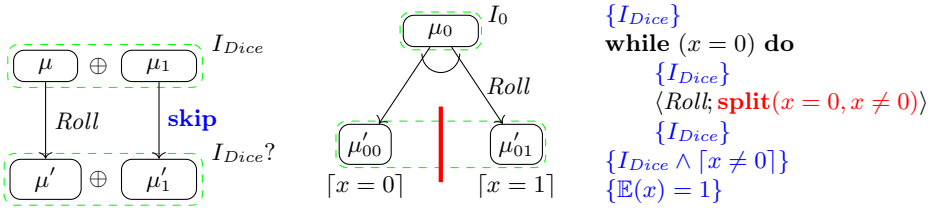
Example: a shared three-sided dice. To see the problem with branch statements more concretely, we consider a simple program $\mathbb{C}_{Dice}$ of n threads, where the code of each thread is *Dice*:

$$\mathit{Dice} \stackrel{\text{def}}{=} \textbf{while } (x = 0) \textbf{ do } \mathit{Roll}, \quad \text{where } \mathit{Roll} \stackrel{\text{def}}{=} (x \stackrel{\$}{:=} \{1 : \tfrac{1}{2} \mid 2x : \tfrac{1}{6} \mid \tfrac{x}{2} : \tfrac{1}{3}\})$$

Here x is a shared variable initialized to 0. The loop body *Roll* is a random assignment, which is short for the atomic probabilistic choice $\langle \langle x := 1 \rangle \oplus_{\frac{1}{2}} (\langle x := 2x \rangle \oplus_{\frac{1}{3}} \langle x := \frac{x}{2} \rangle) \rangle$. That is, the thread atomically rolls a 3-sided dice and updates x according to the outcome: it sets x to 1 with probability $\frac{1}{2}$, doubles x with probability $\frac{1}{6}$ and halves x with probability $\frac{1}{3}$.

We want to verify that $\mathbb{C}_{Dice}$ satisfies the postcondition $\mathbb{E}(x) = 1$. As we explained, to do thread-local reasoning, we first find out the invariant I_{Dice} to model the interference:

$$I_{Dice} \stackrel{\text{def}}{=} I_0 \oplus I_1, \quad \text{where } I_0 \stackrel{\text{def}}{=} \lceil x = 0 \rceil \text{ and } I_1 \stackrel{\text{def}}{=} (\lceil x \neq 0 \rceil \wedge \mathbb{E}(x) = 1)$$



(a) from layer to layer (b) split (c) proof

Fig. 7: Executions of *Dice* and Its Proof with Split

It says, every whole state distribution μ (at every layer of an execution forest) is a mixture $\mu_0 \oplus \mu_1$ (formed by taking μ_0 with arbitrary probability and taking μ_1 with the remaining probability) in which μ_0 and μ_1 satisfy I_0 and I_1 respectively.

To check I_{Dice} is indeed an invariant, one may consider showing that I_{Dice} is preserved by *Roll*. However, even if I_{Dice} is preserved by *Roll* (which is indeed true), it is still unclear whether I_{Dice} is preserved layer by layer. Specifically, after executing *Roll*, we will reach a state distribution whose support contains both the states satisfying $x = 0$ and those satisfying $x \neq 0$. From the former, the thread will enter the next round of the loop; but from the latter, the thread will exit the loop and execute the code after the loop (or **skip** if there is no subsequent code). Consequently, *Roll* may be executed "at the same time" with **skip**, as shown in Fig. 7a. What we need to prove is that I_{Dice} is preserved by a *mixture* of executing *Roll* and **skip** at the same layer.

However, it is difficult to design logic rules to compose the proofs of *Roll* and **skip** for their mixture, because *Roll* as the loop body is actually syntactically sequenced before **skip**, the code after the loop. We face a similar problem as the problem with the **if**-statement, as explained above.

3.5 Our Key Idea: Split

Instead of trying to reason about the mixture of the behaviors of different statements at the whole layer, we *split* the state distribution of the layer, and reason about the different statements separately. In detail, we introduce an auxiliary command $\mathbf{split}(b_1, \ldots, b_k)$. It divides the current state distribution μ into k disjoint parts $\mu_1, \ldots, \mu_k$, such that each smaller distribution μ_i satisfies $\lceil b_i \rceil$ and μ is their mixture $\mu_1 \oplus \ldots \oplus \mu_k$. In our abstract operational semantics the thread *non-deterministically* picks a μ_i and continues its execution. One can instrument the code being verified with proper **split** commands so that each μ_i corresponds to a distinct branch in the control flow. Note that the **split** commands only affects the abstract semantics. In the concrete semantics, **split** has no effect and can be viewed as a no-op.

With split, the invariant I no longer needs to specify the whole layer μ, but instead it specifies only the smaller distributions μ_i generated by split. This I must be preserved by the execution at every μ_i. For instance, if we instrument

split$(b, \neg b)$ before **if** (b) **then** C_1 **else** C_2, then it suffices to prove that I is preserved by the executions of C_1 and C_2 at distributions satisfying $\lceil b \rceil$ and $\lceil \neg b \rceil$ respectively.

Split is physical and irreversible. We do not provide any command to mix back the smaller distributions that result from split. Instead of directly verifying $\vdash_{\text{A}} \{P\}\mathbb{C}\{Q\}$, where $\mathbb{C}$ contains no **split** commands and thus Q holds at the whole leaf layer, we verify $\vdash_{\text{A}} \{P\}\mathbb{C}'\{Q\}$ for $\mathbb{C}'$ that results from instrumenting $\mathbb{C}$ with auxiliary **split** commands. Therefore Q needs to hold at every smaller distribution at the leaf layer. That said, we do provide the following logic rule to convert $\vdash_{\text{A}} \{P\}\mathbb{C}'\{Q\}$ back to $\vdash_{\text{A}} \{P\}\mathbb{C}\{Q\}$:

$$\frac{\vdash_{\text{A}} \{P\}\mathbb{C}'\{Q\} \quad \textbf{closed}(Q)}{\vdash_{\text{A}} \{P\}\textbf{RemoveSplit}(\mathbb{C}')\{Q\}} \ (\textsc{removesplit})$$

Here **RemoveSplit**$(\mathbb{C}')$ removes all the **split** commands from $\mathbb{C}'$, and **closed**(Q) (introduced at the end of Sec. 3.2) allows us to re-establish Q at the mixture of smaller distributions that all satisfy Q. The subscript "A" in the judgement indicates that the reasoning is based on the *abstract* semantics.

Proof for the shared three-sided dice. To verify *Dice*, we split the state distributions so that the states at which the thread enters the next round of the loop and those at which the thread exits the loop are always separate. As such, the invariant I_{Dice} is revised to be a *disjunction*:

$$I_{Dice} \stackrel{\text{def}}{=} I_0 \vee I_1, \quad \text{where } I_0 \stackrel{\text{def}}{=} \lceil x = 0 \rceil \text{ and } I_1 \stackrel{\text{def}}{=} (\lceil x \neq 0 \rceil \wedge \mathbb{E}(x) = 1)$$

In contrast to the earlier $I_0 \oplus I_1$ which holds at a mixture, this new I_{Dice} holds at a state distribution μ satisfying *either* I_0 *or* I_1. If μ satisfies I_0, the thread enters the next round of the loop; otherwise it exits the loop.

We instrument the loop body with the **split** command, as shown in red color in Fig. 7c. This **split** command ensures that the new I_{Dice} is indeed an invariant. As the blue assertions indicate, if I_{Dice} holds before the loop body, which means either I_0 or I_1 holds, then I_{Dice} still holds after atomically executing *Roll* and **split**. In particular, as shown in Fig. 7b, if I_0 holds before the loop body, executing *Roll* gives us a state distribution satisfying $\lceil x = 0 \rceil \oplus \lceil x = 1 \rceil$, and then executing **split**$(x = 0, x \neq 0)$ (see the red vertical bar) results in two separate state distributions μ'_{00} satisfying $\lceil x = 0 \rceil$ and μ'_{01} satisfying $\lceil x = 1 \rceil$. Both μ'_{00} and μ'_{01} satisfy I_{Dice}. The full proof is given in the TR [13].

Logic rules for split and branch statements. Below we introduce our logic rules for **split**, **if**-statements and **while**-loops to show how the split mechanism works.

$$\frac{G \vdash_{\text{SQ}} \{I \wedge P\}C\{(I \wedge Q \wedge \lceil b_1 \rceil) \oplus \cdots \oplus (I \wedge Q \wedge \lceil b_k \rceil)\} \quad \cdots}{R, G, I \vdash \{P\}\langle C \rangle \ \textbf{split}(b_1, \ldots, b_k)\{(Q \wedge \lceil b_1 \rceil) \vee \ldots \vee (Q \wedge \lceil b_k \rceil)\}} \ (\textsc{atom-split})$$

As in the *Dice* example, **split** is usually inserted after and executed atomically with some code $\langle C \rangle$. As such, we provide the command $\langle C \rangle$ **split**$(b_1, \ldots, b_k)$,

which has the same meaning as $\langle C; \mathbf{split}(b_1, \ldots, b_k)\rangle$. The (ATOM-SPLIT) rule requires us to prove the $\vdash_{\text{SQ}}$ judgement, which reasons about C as sequential code, and ensures that the state distribution at the end is a mixture of smaller distributions satisfying $\lceil b_1 \rceil, \ldots, \lceil b_n \rceil$ respectively. Since **split** turns the big distribution into these smaller ones as separate parts, the postcondition of the conclusion is a disjunctive assertion. We can see that split essentially turns $\oplus$ into $\vee$. The disjunction can be the precondition of the subsequent **if** and **while** statements as required by the (COND) and (WHILE) rules below. Here we omit the side conditions which says that the pre/post-conditions are stable with respect to R and I. The definition of rely/guarantee conditions and stability will be explained in Sec. 5.1 and the complete rule will be presented in Sec. 5.2.

$$\frac{P_1 \Rightarrow \lceil b \rceil \quad P_2 \Rightarrow \lceil \neg b \rceil \quad R, G, I \vdash \{P_1\} C_1 \{Q\} \quad R, G, I \vdash \{P_2\} C_2 \{Q\} \quad \cdots}{R, G, I \vdash \{P_1 \vee P_2\} \mathbf{if}\ (b)\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \{Q\}}\ (\text{COND})$$

$$\frac{P_1 \Rightarrow \lceil b \rceil \qquad P_2 \Rightarrow \lceil \neg b \rceil \wedge Q \qquad R, G, I \vdash \{P_1\} C \{P_1 \vee P_2\} \qquad \cdots}{R, G, I \vdash \{P_1 \vee P_2\} \mathbf{while}\ (b)\ \mathbf{do}\ C \{Q\}}\ (\text{WHILE})$$

Our (COND) rule assumes that, before the **if**-statement, the state distributions have already been split into smaller distributions for executing the **then**- and **else**-branches separately. Therefore, the precondition is supposed to be the disjunction $P_1 \vee P_2$, where $P_1 \Rightarrow \lceil b \rceil$ and $P_2 \Rightarrow \lceil \neg b \rceil$. Recall that $\lceil b \rceil$ says b holds with probability 1, i.e., all the states in the support of the distribution satisfy b. So, $\lceil b \rceil \vee \lceil \neg b \rceil$ is not implied by $\lceil b \vee \neg b \rceil$. The latter holds always, but for the former to hold, we must do split first. Then the branches can be verified independently, as we do in classical Hoare logic.

Similarly, in the (WHILE) rule, the loop invariant is the disjunction $P_1 \vee P_2$. Resulting from a split, the part satisfying P_1 ensures that the loop always continues with its next round since $P_1 \Rightarrow \lceil b \rceil$, while the part satisfying P_2 terminates the loop as $P_2 \Rightarrow \lceil \neg b \rceil$. If the value of b is probabilistic and can be modified by the code before the loop and by the loop body C, one need to insert **split** before the loop *and* inside the loop body C, so that $P_1 \vee P_2$ holds before every round of the loop.

4 The Programming Language

The syntax of the language is defined in Fig. 8. The whole program $\mathbb{C}$ consists of n sequential threads. The statements C of each thread are mostly standard. The *atomic statements* $\langle C \rangle$ and *the probabilistic choices* $\langle C_1 \rangle \oplus_p \langle C_2 \rangle$ are explained in Sec. 3. For verification purposes, we also append the atomic statements with split statements to get $(\langle C \rangle\ sp)$ where sp is in the form of $\mathbf{split}(b_1, \ldots, b_k)$.

Below we give two operational semantics to the language. The concrete one follows the standard interleaving semantics and models program steps as *probabilistic* transitions over program states. *The split statements are ignored in this semantics.* That is, they are viewed as annotations for verification only and have no operational effects.

$$
\begin{array}{rrl}
(\mathit{Nat}) & n, k & \in \mathbb{N} \qquad (\mathit{Real}) \quad p, r \in \mathbb{R} \qquad (\mathit{PVar}) \quad x \in \mathit{String} \\
(\mathit{Expr}) & e & ::= n \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid \ldots \\
(\mathit{Bexp}) & b & ::= \text{true} \mid \text{false} \mid e_1 < e_2 \mid e_1 = e_2 \mid e_1 \leq e_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \ldots \\
(\mathit{SplitInstr}) & sp & ::= \mathbf{split}(b_1, \ldots, b_k) \\
(\mathit{Stmt}) & C & ::= \mathbf{skip} \mid x := e \mid C_1; C_2 \mid \mathbf{if}\ (b)\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \mid \mathbf{while}\ (b)\ \mathbf{do}\ C \\
 & & \quad \mid \langle C \rangle \mid \langle C \rangle\ sp \mid \langle C_1 \rangle \oplus_p \langle C_2 \rangle \\
(\mathit{Prog}) & \mathbb{C} & ::= C_1 \parallel \cdots \parallel C_n
\end{array}
$$

Fig. 8: The Programming Language

Thread IDs, schedules, states and states distributions:

$$
\begin{array}{ll}
(\mathit{ThreadId})\ t \in \mathbb{N}_+ & (\mathit{Schedule})\ \varphi ::= t :: \varphi \quad \text{(coinductive)} \\
(\mathit{State})\ \sigma \in \mathit{PVar} \to \mathbb{R} & (\mathit{DState})\ \mu \in \mathbb{D}_{\mathit{State}}
\end{array}
$$

Global transitions: $(\mathbb{C}, \sigma) \xrightarrow[t]{p} (\mathbb{C}', \sigma')$

$$
\frac{(C_t, \sigma) \xrightarrow{p} (C_t', \sigma')}{(C_1 \parallel \cdots \parallel C_t \parallel \cdots \parallel C_n, \sigma) \xrightarrow[t]{p} (C_1 \parallel \cdots \parallel C_t' \parallel \cdots \parallel C_n, \sigma')}
$$

Thread-local transitions: $(C, \sigma) \xrightarrow{p} (C', \sigma')$

$$
\frac{[\![e]\!]_\sigma = n}{(x := e, \sigma) \xrightarrow{1} (\mathbf{skip}, \sigma\{x \rightsquigarrow n\})} \qquad \frac{}{(\mathbf{skip}, \sigma) \xrightarrow{1} (\mathbf{skip}, \sigma)}
$$

$$
\frac{C_1 \neq \mathbf{skip} \quad (C_1, \sigma) \xrightarrow{p} (C_1', \sigma')}{(C_1; C_2, \sigma) \xrightarrow{p} (C_1'; C_2, \sigma')} \qquad \frac{}{(\mathbf{skip}; C_2, \sigma) \xrightarrow{1} (C_2, \sigma)}
$$

$$
\frac{}{(\langle C_1 \rangle \oplus_p \langle C_2 \rangle, \sigma) \xrightarrow{p} (\langle C_1 \rangle, \sigma)} \qquad \frac{}{(\langle C_1 \rangle \oplus_p \langle C_2 \rangle, \sigma) \xrightarrow{1-p} (\langle C_2 \rangle, \sigma)}
$$

$$
\frac{\exists k. \forall n \geq k.\, (C, \sigma) \xrightarrow{p}{}^{n} (\mathbf{skip}, \sigma')}{(\langle C \rangle, \sigma) \xrightarrow{p} (\mathbf{skip}, \sigma')} \qquad \frac{(\langle C \rangle, \sigma) \xrightarrow{p} (\mathbf{skip}, \sigma')}{(\langle C \rangle\ \mathbf{split}(b_1, \ldots, b_k), \sigma) \xrightarrow{p} (\mathbf{skip}, \sigma')}
$$

Fig. 9: Concrete Operational Semantics

The abstract semantics models program steps as transitions over distributions of program configurations. We also assign operational semantics to **split** statements. We prove that Hoare-triples valid in the abstract semantics are also valid in the concrete semantics (Thm 1 below).

4.1 Concrete Operational Semantics

We show selected semantics rules in Fig. 9 and give the full set of rules in the TR [13]. The single-step transition of the whole program is defined through the thread-local transitions. Each step is decorated with a p, the probability that the step may occur. For most thread-local transitions except the probabilistic choices and atomic statements, p is simply 1. Note that we allow the **skip** command at the end of execution to stutter with probability 1, but it cannot stutter if it is sequenced before some C. That is, "$\mathbf{skip}; C$" can only step to C. $\langle C_1 \rangle \oplus_p \langle C_2 \rangle$

chooses to execute the left or right branches, with probability p and $1-p$, respectively. The atomic statement $\langle C\rangle$ is always done in one step, no matter how complicated C is. We assume C in the atomic statement never contains **while**-loops, so it always terminates in a bounded number of steps. Note that the need of atomicity of the branches in $\langle C_1\rangle \oplus_p \langle C_2\rangle$ is not overly idealistic, because we mainly use $\langle C_1\rangle \oplus_p \langle C_2\rangle$ to encode a random assignment, thus C_1 and C_2 themselves may correspond to single instructions at the machine level anyway (in this case, the atomic wrappers $\langle\cdot\rangle$ are unnecessary). In the proofs of algorithms, we may insert auxiliary statements (a.k.a. ghost code) to be executed with the probabilistic choice together in one step. This is actually the only case when C_1 or C_2 is non-atomic and needs to be wrapped by $\langle\cdot\rangle$. The more general form of $C_1 \oplus_p C_2$ can be encoded as $\langle x := \text{true}\rangle \oplus_p \langle x := \text{false}\rangle; \textbf{if } (x) \textbf{ then } C_1 \textbf{ else } C_2$.

Before giving semantics to $\langle C\rangle$, we first introduce the n-step thread-local transition, represented as $(C,\sigma) \xrightarrow{p}{}^{n} (C',\sigma')$. Informally, if there is only one n-step execution path from (C,σ) to (C',σ'), the probability p in $(C,\sigma) \xrightarrow{p}{}^{n} (C',\sigma')$ is the product of the probability of every step on the path. If there are more than one execution paths, we need to sum up the probabilities of all the paths. We present the formal definition of the n-step thread-local transition and an illustrative example in the TR [13].

Then the operational semantics rule for $\langle C\rangle$ says it finishes the execution of C in one step (that is, the execution of C cannot be interrupted by other threads). Note that $\langle C\rangle$ may lead to different states with different probabilities, since C may contain probabilistic choices.

The multi-step transition $((\mathbb{C},\sigma) \xrightarrow[\varphi]{p}{}^{n} (\mathbb{C}'',\sigma''))$ of the whole program $\mathbb{C}$ under the schedule φ is similar to the multi-step thread-local transitions. The schedule φ is an infinite sequence of thread IDs. It decides which thread t is to be executed next. The accumulated probability of an n-step transition is the *sum* of the probability of every possible execution path.

Below we define $[\![\mathbb{C}]\!]_\varphi$ as a function that maps an *initial state* σ to a sub-distribution of *final states*. We also lift the function to the distribution μ of the initial states.

$$[\![\mathbb{C}]\!]_\varphi(\sigma) \stackrel{\text{def}}{=} \lambda\sigma'.\ \lim \vec{p}_{\sigma'}, \quad \text{where } \forall n.\, (\mathbb{C},\sigma) \xrightarrow[\varphi]{\vec{p}_{\sigma'}[n]}{}^{n} (\textbf{skip} \parallel \cdots \parallel \textbf{skip}, \sigma')$$

$$[\![\mathbb{C}]\!]_\varphi(\mu) \stackrel{\text{def}}{=} \mathbb{E}_{\sigma\sim\mu}\{[\![\mathbb{C}]\!]_\varphi(\sigma)\} \qquad \text{(see Def. 7 for the expected sub-distribution)}$$

Here $\vec{p}_{\sigma'}$ is an infinite sequence of probabilities and $\vec{p}_{\sigma'}[n]$ is the n-th element of the sequence. Note $\lim \vec{p}_{\sigma'}$ always exists as we can prove $\vec{p}_{\sigma'}$ always converges.

Then we can give a simple definition of the partial correctness of $\mathbb{C}$ with respect to the precondition P and the postcondition Q, which are assertions over state distributions and are defined in Sec. 5.1.

Definition 9. $\models \{P\}\mathbb{C}\{Q\}$ iff, for all μ and φ, if $\mu \models P$, and $|[\![\mathbb{C}]\!]_\varphi(\mu)| = 1$, then $[\![\mathbb{C}]\!]_\varphi(\mu) \models Q$.

The premise $|[\![\mathbb{C}]\!]_\varphi(\mu)| = 1$ requires the execution of $\mathbb{C}$ (with the schedule φ and the initial state distribution μ) *terminates with probability* 1.

$$
\begin{array}{rcl}
W & \in & \mathbb{D}_{Prog \times State} \qquad\qquad W|_b \stackrel{\text{def}}{=} W|_{\lambda(\mathbb{C},\sigma).\sigma \models b} \\
\delta(\mathbb{C}) & \stackrel{\text{def}}{=} & \lambda \mathbb{C}_1. \begin{cases} 1, & \text{if } \mathbb{C}_1 = \mathbb{C} \\ 0, & \text{otherwise} \end{cases} \\
\mathit{init}(\mathbb{C}, \mu) & \stackrel{\text{def}}{=} & \delta(\mathbb{C}) \otimes \mu \qquad \text{(see Eqn. (4) for the definition of } \otimes) \\
\mathit{nextsplit}(C) & \stackrel{\text{def}}{=} & \begin{cases} \textbf{split}(b_1, \ldots, b_k), & \text{if } C = \langle C_1 \rangle \ \textbf{split}(b_1, \ldots, b_k) \\ \mathit{nextsplit}(C_1), & \text{if } C = C_1; C_2 \\ \textbf{split}(\text{true}), & \text{otherwise} \end{cases} \\
\mathit{nextsplit}(W, t) & \stackrel{\text{def}}{=} & \{\mathit{nextsplit}(C_t) \mid (C_1 \parallel \cdots \parallel C_n, \sigma) \in \mathit{supp}(W)\} \\
W \overset{t}{\rightsquigarrow} W' & & \text{iff } W' = \lambda(\mathbb{C}', \sigma'). \sum_{\mathbb{C},\sigma} \{p \cdot W(\mathbb{C}, \sigma) \mid (\mathbb{C}, \sigma) \xrightarrow[t]{p} (\mathbb{C}', \sigma')\}
\end{array}
$$

$$
\frac{W \overset{t}{\rightsquigarrow} W' \quad \mathit{nextsplit}(W, t) = \{\textbf{split}(b_1, \ldots, b_k)\} \quad W'|_{b_i} = W''}{W \overset{t}{\hookrightarrow} W''}
$$

$$
\frac{W \overset{t}{\rightsquigarrow} W' \quad \#\mathit{nextsplit}(W, t) > 1}{W \overset{t}{\hookrightarrow} W'}
$$

Fig. 10: Abstract Operational Semantics

4.2 Abstract Operational Semantics

The abstract semantics, shown in Fig. 10, models each step as a transition between distributions W of the whole program configurations $(\mathbb{C}, \sigma)$. Also we give semantics to **split** statements.

Below we use $\mathit{nextsplit}(W, t)$ to represent the set consisting of the next **split** statements to be executed in the thread t of the program configurations in $\mathit{supp}(W)$. The next **split** statement of the thread t is sp if the next statement to be executed is in the form of $\langle C \rangle\ sp$, otherwise the next split is defined as $\textbf{split}(\text{true})$. Throughout this paper, we assume all the splits $\textbf{split}(b_1, \ldots, b_k)$ satisfy the following validity check, which says for any state there is always one and only one b_i that holds.

Definition 10. A split statement is valid, i.e., $\textbf{validsplit}(\textbf{split}(b_1, \ldots, b_k))$ holds, if and only if for any state σ, $\forall i, j.\ i \neq j \implies \sigma \models \neg(b_i \wedge b_j)$ and $\sigma \models b_1 \vee \ldots \vee b_k$.

The transition $W \overset{t}{\hookrightarrow} W''$ is done in two steps. First we make the transition $W \overset{t}{\rightsquigarrow} W'$ based on the concrete semantics, without considering splits. Then the splits in $\mathit{nextsplit}(W, t)$ are executed. We expect $\mathit{nextsplit}(W, t)$ to be a singleton set, i.e., threads t in different program configurations in $\mathit{supp}(W)$ all have the same subsequent **split** statement. We non-deterministically pick a b_i from $b_1 \ldots b_k$, and let W'' be the filtered distribution $W'|_{b_i}$ (see Fig. 10 and Eqn. (2) for the definition of $W|_b$). If the **split** statement is $\textbf{split}(\text{true})$, we know W'' is the same as W'. If $\mathit{nextsplit}(W, t)$ contains more than one **split** statements, then we view the program as inappropriately instrumented. In this case we ignore all the split statements in $\mathit{nextsplit}(W, t)$ and let W'' be W'.

Figure 11 illustrates the execution. The dashed arrows represent state transitions in the concrete semantics, while the solid arrows represent the transitions $W \overset{t}{\rightsquigarrow} W'$ in the abstract semantics. Like before, we use different colors to represent actions of different threads. The vertical bars represent splits. The solid arrow and the split together correspond to the transition $W \overset{t}{\hookrightarrow} W''$. The branching shown by the two solid red arrows reflects the *non-deterministic* choice of the cases of the split.

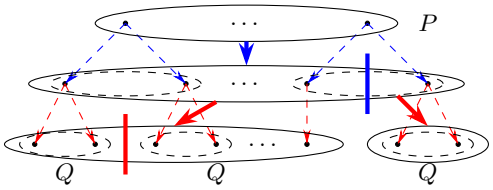


Fig. 11: Illustration of $\models_{\text{A}} \{P\}\mathbb{C}\{Q\}$

Before giving the partial correctness under the abstract semantics, we first define the termination of W_0 in Def. 11: if the execution sequence of W_0 under the abstract semantics converges with the limit W, we say W_0 terminates at W.

Definition 11 (Termination of W). Given W_0 and a schedule φ. We say W_0 *terminates at* W *under the schedule* φ, represented as $W_0 \Downarrow_\varphi W$, if and only if there is an infinite sequence $\vec{W}$ such that $\mathbf{History}(W_0, \varphi, \vec{W})$, $\lim \vec{W} = W$ and $W^{(Prog)}(\mathbf{skip} \parallel \cdots \parallel \mathbf{skip}) = 1$.

Here $\mathbf{History}(W_0, \varphi, \vec{W})$ says that $\vec{W}$ is a possibly infinite sequence $W_0, W_1, \ldots$ where $W_i \overset{\varphi[i]}{\hookrightarrow} W_{i+1}$ for every i. The formal definition of **History** can be found in the TR [13]. The limit $(\lim \vec{W})$ is defined by Def. 6. The projection of W over code $(W^{(Prog)})$ and state $(W^{(State)})$ are defined by Eqn. (5).

Next we define the partial correctness under the abstract semantics, $\models_{\text{A}} \{P\}\mathbb{C}\{Q\}$. The initial distribution of program configurations is $init(\mathbb{C}, \mu)$. As defined in Fig. 10, $init(\mathbb{C}, \mu)$ says the initial program is always $\mathbb{C}$ and the state distribution is μ. Figure 11 illustrates the meaning of $\models_{\text{A}} \{P\}\mathbb{C}\{Q\}$: if P holds over the initial distribution, Q must hold over *every* final distribution. Theorem 1 shows that the partial correctness in the abstract semantics implies the partial correctness in the concrete semantics when the postcondition is closed. Below we develop our program logic based on this abstract semantics.

Definition 12. $\models_{\text{A}} \{P\}\mathbb{C}\{Q\}$ iff for all μ, if $\mu \models P$, then for all φ and W, if $init(\mathbb{C}, \mu) \Downarrow_\varphi W$, then $W^{(State)} \models Q$.

Theorem 1. For all $P, \mathbb{C}, Q$, if $\models_{\text{A}} \{P\}\mathbb{C}\{Q\}$ and $\mathbf{closed}(Q)$, then $\models \{P\}\mathbb{C}\{Q\}$.

5 The Program Logic

We present the assertion language and the logic rules in this section.

$$
\begin{array}{rrl}
(\textit{Assertion}) & \mathbf{p}, \mathbf{q} & ::= b \mid \neg \mathbf{q} \mid \mathbf{q}_1 \wedge \mathbf{q}_2 \mid \mathbf{q}_1 \vee \mathbf{q}_2 \mid \forall X.\mathbf{q} \mid \exists X.\mathbf{q} \mid \ldots \\
(\textit{Pexp}) & \xi & ::= r \mid \mathbb{E}(e) \mid \mathbf{Pr}(\mathbf{q}) \mid \xi_1 + \xi_2 \mid \xi_1 - \xi_2 \mid \xi_1 * \xi_2 \mid \ldots \\
(\textit{PAssertion}) & P, Q, M, I & ::= \lceil \mathbf{q} \rceil \mid \xi_1 < \xi_2 \mid \xi_1 = \xi_2 \mid \xi_1 \leq \xi_2 \mid \neg Q \mid Q_1 \wedge Q_2 \mid Q_1 \vee Q_2 \\
 & & \quad \mid \forall X.Q \mid \exists X.Q \mid Q_1 \oplus_p Q_2 \mid Q_1 \oplus Q_2 \mid \ldots \\
(\textit{Action}) & R, G & ::= \mathbf{p} \ltimes \mathbf{q} \mid [\mathbf{q}] \mid \neg R \mid R_1 \wedge R_2 \mid R_1 \vee R_2 \mid \forall X.R \mid \exists X.R \mid R_1 \circ R_2 \mid \ldots
\end{array}
$$

Fig. 12: The Assertion Language

Evaluation of probabilistic expressions:

$$[\![\mathbb{E}(e)]\!]_\mu \stackrel{\text{def}}{=} \mathbb{E}_{\sigma\sim\mu}[[\![e]\!]_\sigma] \qquad [\![\mathbf{Pr}(\mathbf{q})]\!]_\mu \stackrel{\text{def}}{=} \mathbf{Pr}_{\sigma\sim\mu}[\sigma \models \mathbf{q}]$$

Semantics of probabilistic assertions:

$\mu \models \lceil \mathbf{q} \rceil$ iff for all $\sigma \in \mathit{supp}(\mu)$, $\sigma \models \mathbf{q}$

$\mu \models Q_1 \oplus_p Q_2$ iff $p = 1$ and $\mu \models Q_1$, or $p = 0$ and $\mu \models Q_2$, or $0 < p < 1$ and there exists μ_1 and μ_2 such that $\mu = \mu_1 \oplus_p \mu_2$, $\mu_1 \models Q_1$ and $\mu_2 \models Q_2$

$\mu \models Q_1 \oplus Q_2$ iff there exists p such that $\mu \models Q_1 \oplus_p Q_2$

Fig. 13: Semantics of Assertions

5.1 The Assertion Language

We show the syntax of assertions in Fig. 12 and their semantics in Fig. 13. We use $\mathbf{p}$ and $\mathbf{q}$ to represent classical assertions over states, and P, Q and I for *probabilistic assertions* over state distributions. We also use ξ to denote *probabilistic expressions* such as the expected value of an arithmetic expression or the probability of a classical assertion. The expression ξ evaluates to a real number under the state distribution μ, represented as $[\![\xi]\!]_\mu$. $\mathbb{E}(e)$ evaluates to the expected value of $[\![e]\!]_\sigma$ (where $\sigma \in \mathit{supp}(\mu)$). $\mathbf{Pr}(\mathbf{q})$ evaluates to the probability of $\sigma \models \mathbf{q}$ (where $\sigma \in \mathit{supp}(\mu)$). The key definitions of expected values and probability of assertions are shown in Eqn. (1).

The assertion $\lceil \mathbf{q} \rceil$ lifts the state assertion $\mathbf{q}$ to a probabilistic assertion. It says $\mathbf{q}$ holds on all states in the support of the state distribution. The assertion $P \oplus_p Q$ holds at μ, if μ is a *mixture* of two distributions μ_0 and μ_1, which are associated with probabilities p and $1-p$, and satisfy P and Q respectively. $Q_1 \oplus Q_2$ says there exists p such that $Q_1 \oplus_p Q_2$ holds. The semantics of $\forall X.Q$ and $\exists X.Q$ are given in the TR [13]. Throughout this paper, we use capital letters X to indicate that X is a logical variable and lowercase letters x to indicate that x is a program variable. We define **true** as a syntactic sugar of $\lceil \text{true} \rceil$ which holds on all state distributions.

Actions R and G are assertions over state transitions. Their semantics, $(\sigma, \sigma') \models R$, is the same as that in classical (non-probabilistic) rely-guarantee logics. We use $[\![R]\!]$ to denote the set of state transitions that satisfy R.

Stability We define the stability of a probabilistic assertion Q with respect to the environment interference (specified by I and R) in Fig. 14. We first define

$$\mu \overset{R}{\mapsto} \mu' \quad \text{iff } \exists \theta \in \mathcal{P}(State \times State).\, \theta \subseteq [\![R]\!] \wedge supp(\mu) = dom(\theta) \wedge supp(\mu') = range(\theta)$$

$$\mu \underset{I}{\overset{R}{\rightarrowtail}} \mu'' \quad \text{iff } \mu \models I \wedge (\exists \mu'.\, \mu \overset{R}{\mapsto} \mu' \wedge supp(\mu'') \subseteq supp(\mu')) \wedge \mu'' \models I$$

$$\mathbf{Sta}(Q, R, I) \text{ iff } \forall \mu, \mu'.\, \mu \models Q \wedge \mu \underset{I}{\overset{R}{\rightarrowtail}} \mu' \implies \mu' \models Q$$

Fig. 14: Stability

$\mu \underset{I}{\overset{R}{\rightarrowtail}} \mu''$ to describe that the current state distribution is changed from μ to μ'' due to the environment interference. As we can see in the abstract operational semantics, every transition made by a thread is done in two steps. The first step is normal execution without splits and the second step is the execution of **split**. Similarly, we model the execution of the environment in two steps. The first step is $\mu \overset{R}{\mapsto} \mu'$. It requires us to find a set θ of state transitions allowed by R (i.e. $\theta \subseteq [\![R]\!]$), such that θ transforms the states of $supp(\mu)$ to those of $supp(\mu')$. The second step is the execution of **split** statements by the environment. The condition $supp(\mu'') \subseteq supp(\mu')$ abstracts the behaviors of **split**. In addition, the environment needs to preserve the invariant I, so $\mu \models I \wedge \mu'' \models I$. Then we can give a simple definition of $\mathbf{Sta}(Q, R, I)$ in Fig. 14.

In general, it is not easy to prove the stability of a probabilistic assertion with respect to classical rely conditions. But in practice, the thread-local pre/post-conditions and intermediate assertions P are usually "non-probabilistic", in the form of $\lceil b_1 \rceil \vee \ldots \vee \lceil b_n \rceil$. This is because the probabilistic information is often about the shared resource and has already been specified by the global invariant I. For such P, proving stability $\mathbf{Sta}(P, R, I)$ is not much harder than proving stability in the classical rely-guarantee reasoning. We give some rules to syntactically proving $\mathbf{Sta}(P, R, I)$ in the TR [13].

Closed Assertions As explained in Sec. 3.5, we need the postcondition of the whole program to be closed for applying split. $\mathbf{closed}(Q)$ means that the mixture of any (maybe countably infinite) number of state distributions satisfies Q if each of them satisfies Q.

Definition 13. An assertion Q is closed, i.e., $\mathbf{closed}(Q)$ holds, if and only if, for all $\nu \in \mathbb{D}_{\mathbb{D}_{State}}$, if $\mu \models Q$ holds for all $\mu \in supp(\nu)$, then $\overline{\nu} \models Q$ (see Eqn. (8) for the definition of $\overline{\nu}$).

Many assertions are closed, such as $\lceil x = 1 \rceil$, $\mathbf{Pr}(y > 2) = 0.5$, $\lceil x = 0 \rceil \oplus \lceil x = 1 \rceil$. We give syntactic rules in the TR [13] to prove closedness of assertions. There do exist non-closed assertions, such as $\lceil x = 1 \rceil \vee \lceil x = 2 \rceil$ and $\mathbf{Pr}(x = 0) \neq 0.5$. In this work, we focus on the class of randomized algorithms whose correctness is about the bound of the probability of a random event or the expected value of a random variable. For this kind of algorithms, our syntactic rules for closedness are useful enough.

Limit-Closed Assertions To verify almost surely terminating programs, we require the invariant I and the postconditions of all threads are limit-closed assertions. Below we define limit-closed assertions (see Def. 6 for the definition of $\lim \vec{\mu}$).

Definition 14. An assertion Q is limit-closed, i.e., **lclosed**(Q) holds, if and only if, for all infinite sequences $\vec{\mu}$, if $\lim \vec{\mu} = \mu$, and $\vec{\mu}[n] \models Q$ holds for all n, then $\mu \models Q$.

We also give syntactic rules in the TR [13] to prove that an assertion is limit-closed. They are similar to those for closedness and thus are also useful in verifying algorithms whose correctness is about the bound of the probability of a random event or the expected value of a random variable.

5.2 Inference Rules

Our inference rules are organized into three layers for the whole program, the thread local reasoning, and sequential reasoning, as shown in Fig. 15. The top-level judgement for the whole program is in the form of $\vdash_{\text{A}} \{P\}\mathbb{C}\{Q\}$ where "A" means abstract. One can use the parallel composition rule (PAR) to decompose the verification of concurrent programs into the verification of each thread. The judgement for thread-local reasoning is in the form of $R, G, I \vdash \{P\}C\{Q\}$ where R and G are rely/guarantee conditions and I is the layer invariant. To verify atomic blocks, one can use the (ATOM) and (ATOM-SPLIT) rules to apply sequential reasoning to the code in the atomic blocks. The judgement for sequential reasoning is in the form of $\vdash_{\text{SQ}} \{P\}C\{Q\}$ where "SQ" means sequential.

Whole-Program Rules The top-level rules are used to verify whole programs. The judgement is in the form of $\vdash_{\text{A}} \{P\}\mathbb{C}\{Q\}$. Here P and Q are probabilistic assertions, which specify the initial state distributions and the terminating state distributions respectively.

The parallel composition rule (PAR) is (mostly) standard. The invariant I and the postcondition of each thread $Q_1, \ldots, Q_n$ are required to be limit-closed assertions, which ensures that the limit state distribution of the infinite sequence produced by $\mathbb{C}$ under the abstract operational semantics satisfies I and $Q_1, \ldots, Q_n$.

The (LAZYCOIN) rule is used to verify probabilistic choices. Note that the execution of $\langle C_1\rangle \oplus_p \langle C_2\rangle$ is *not* atomic, and its two steps (i.e. the coin flip and the execution of $\langle C_1\rangle$ or $\langle C_2\rangle$) can interleave with the environment steps. The (LAZYCOIN) rule allows us to verify **lazycoin**$(\mathbb{C})$ instead of $\mathbb{C}$, where **lazycoin**$(\mathbb{C})$ replaces every $\langle C_1\rangle \oplus_p \langle C_2\rangle$ in $\mathbb{C}$ with **skip**; $\langle\langle C_1\rangle \oplus_p \langle C_2\rangle\rangle$. We can view **lazycoin** as a transformation that defers the coin flip step to be executed with $\langle C_1\rangle$ or $\langle C_2\rangle$ together. This transformation is sound because, in the OA model, the scheduler and the environment threads should not be aware of the outcome of the coin flip, so we can soundly swap the coin-flip step and the environment steps, and reason about the atomic probabilistic choice $\langle\langle C_1\rangle \oplus_p \langle C_2\rangle\rangle$ instead. The extra **skip** is to ensure that the new code has the same number of steps as the non-atomic

Whole program rules: $\vdash_{\text{A}} \{P\}\mathbb{C}\{Q\}$

$$\frac{\forall i.\ R_i, G_i, I \vdash \{P_i\}C_i\{Q_i\} \qquad \forall i, j.\ i \neq j \implies (G_i \Rightarrow R_j) \qquad P \Rightarrow I \wedge P_1 \wedge \cdots \wedge P_n \qquad I \wedge Q_1 \wedge \cdots \wedge Q_n \Rightarrow Q \qquad \mathbf{lclosed}(\{I, Q_1, \ldots, Q_n\})}{\vdash_{\text{A}} \{P\}C_1 \parallel \cdots \parallel C_n\{Q\}}\ (\textsc{par})$$

$$\frac{\vdash_{\text{A}} \{P\}\mathbb{C}\{Q\} \quad \mathbf{closed}(Q)}{\vdash_{\text{A}} \{P\}\mathbf{RemoveSplit}(\mathbb{C})\{Q\}}\ (\textsc{removesplit}) \qquad \frac{\vdash_{\text{A}} \{P\}\mathbf{lazycoin}(\mathbb{C})\{Q\}}{\vdash_{\text{A}} \{P\}\mathbb{C}\{Q\}}\ (\textsc{lazycoin})$$

Thread-local rules: $R, G, I \vdash \{P\}C\{Q\}$

$$\frac{G \vdash_{\text{SQ}} \{I \wedge P\}C\{I \wedge Q\} \qquad \mathbf{Sta}(\{P, Q\}, R, I)}{R, G, I \vdash \{P\}\langle C\rangle\{Q\}}\ (\textsc{atom}) \qquad \frac{R, G, I \vdash \{P\}C_1\{M\} \qquad R, G, I \vdash \{M\}C_2\{Q\}}{R, G, I \vdash \{P\}C_1; C_2\{Q\}}\ (\textsc{seq})$$

$$\frac{G \vdash_{\text{SQ}} \{I \wedge P\}C\{(I \wedge Q \wedge \lceil b_1 \rceil) \oplus \cdots \oplus (I \wedge Q \wedge \lceil b_k \rceil)\} \qquad \mathbf{Sta}(\{P, Q \wedge (\lceil b_1 \rceil \vee \cdots \vee \lceil b_k \rceil)\}, R, I)}{R, G, I \vdash \{P\}\langle C\rangle\ \mathbf{split}(b_1, \ldots, b_k)\{(Q \wedge \lceil b_1 \rceil) \vee \ldots \vee (Q \wedge \lceil b_k \rceil)\}}\ (\textsc{atom-split})$$

$$\frac{P_1 \Rightarrow \lceil b \rceil \qquad P_2 \Rightarrow \lceil \neg b \rceil \qquad \mathbf{Sta}(P_1 \vee P_2, R, I) \qquad R, G, I \vdash \{P_1\}C_1\{Q\} \qquad R, G, I \vdash \{P_2\}C_2\{Q\}}{R, G, I \vdash \{P_1 \vee P_2\}\mathbf{if}\ (b)\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\{Q\}}\ (\textsc{cond})$$

$$\frac{P_1 \Rightarrow \lceil b \rceil \quad P_2 \Rightarrow \lceil \neg b \rceil \wedge Q \quad R, G, I \vdash \{P_1\}C\{P_1 \vee P_2\} \quad \mathbf{Sta}(\{P_1 \vee P_2, Q\}, R, I)}{R, G, I \vdash \{P_1 \vee P_2\}\mathbf{while}\ (b)\ \mathbf{do}\ C\{Q\}}\ (\textsc{while})$$

Fig. 15: Selected Logic Rules

$\langle C_1\rangle \oplus_p \langle C_2\rangle$, and thus to ensure that $\mathbf{lazycoin}(\mathbb{C})$ and $\mathbb{C}$ generate the same behaviors in the OA model. Note that (LAZYCOIN) is *unsound* in the SA model.

The (REMOVESPLIT) rule has been explained in Sec. 3. We also support the standard consequence rule, conjunction rule and disjunction rule for whole programs, which are shown in the TR [13].

Thread-Local Rules The thread-local judgement is in the form of $R, G, I \vdash \{P\}C\{Q\}$. The rely/guarantee conditions R and G are non-probabilistic and their meaning are the same as in the traditional rely-guarantee reasoning. The invariant I specifies the probabilistic property that is preserved by both the thread and its environment at every layer. The rely/guarantee conditions need to be reflexive in well-formed thread-local judgements.

To verify $\langle C\rangle$, the (ATOM) rule asks one to verify C as sequential code, and requires I is preserved at the end if it holds at the beginning, and the whole state transitions resulting from the sequential execution C satisfy the guarantee G. The pre/post-conditions need to be stable with respect to R and I. We use $\mathbf{Sta}(\{P, Q\}, R, I)$ as a shorthand for $\mathbf{Sta}(P, R, I) \wedge \mathbf{Sta}(Q, R, I)$. Similar representations are used in the remaining part of the paper.

Our (SEQ) rule for sequential composition is standard. The (ATOM-SPLIT), (COND) and (WHILE) rules have been explained in Sec. 3.5. Note that (ATOM-

SPLIT) cannot be replaced by (ATOM), since only **split** can turn $\oplus$ into $\vee$ (see the first premise and conclusion's postconditions in (ATOM-SPLIT)).

Sequential Rules The judgement for sequential rules is in the form of $G \vdash_{\text{SQ}} \{P\}C\{Q\}$. Note that the guarantee G does not specify the state transition of every single step of C. Instead it specifies the state transitions from initial states to the corresponding final states at the end of C. The rules for sequential reasoning are simple extensions of those in [6] and are presented in the TR [13].

Soundness The following theorem shows that our logic is sound with respect to the abstract operational semantics, where $\models_{\text{A}} \{P\}\mathbb{C}\{Q\}$ is given in Def. 12.

Theorem 2. *For all* $P, \mathbb{C}, Q$*, if* $\vdash_{\text{A}} \{P\}\mathbb{C}\{Q\}$*, then* $\models_{\text{A}} \{P\}\mathbb{C}\{Q\}$.

6 Case Study: Conciliator

As introduced in Sec. 1, Chor et al. [12] give a probabilistic-write based conciliator for *probabilistic agreement* between n threads, each thread i executing C_i below, where s is a shared variable and y_i is the local variable for thread i that records its return value.

$$C_i \stackrel{\text{def}}{=} (\textbf{while } (s = 0) \textbf{ do } \langle s := i\rangle \oplus_p \langle \textbf{skip}\rangle)\,; y_i := s$$

We want to prove $\{\lceil s = 0\rceil\}C_1 \parallel \cdots \parallel C_n\{\textbf{Pr}(y_1 = \cdots = y_n) \geq (1-p)^{n-1}\}$. Intuitively the postcondition holds because, when there is exactly one thread i which succeeds in writing to s, all threads will return i. This ideal case happens with probability no less than $(1-p)^{n-1}$ in OA, because (i) for the program to terminate, at least one thread has updated s, and (ii) after the first update to s, each of the other $n-1$ threads has at most one chance to update s, and such an update happens with probability no more than $1-p$. Note that this algorithm does *not* work in SA, where different threads can be scheduled for different outcomes of coin flips. For example, a strong adversary may behave as follows: It first non-deterministically selects a thread and keeps scheduling it until it flips heads. It then selects another thread and schedules it in the same manner, until all threads have flipped heads. After that, it schedules each thread for two consecutive steps, so that each returns its own number. In this case, the probability of agreement is 0.

To formulate the intuition, we introduce a shared auxiliary variable c that counts how many threads have updated s and insert the auxiliary code $c := c+1$ which is executed atomically with $s := i$. We also introduce flag variables d_i to formalize the "at most one chance" update to s. When d_i is set, it means thread i can no longer update s. We insert the auxiliary code $\mathit{SetFlag}_i$ to set d_i at the proper time. At the whole-program level, we apply (LAZYCOIN) and (REMOVESPLIT) to wrap the probabilistic choice in an atomic block, and to instrument $\textbf{split}(s = 0, s \neq 0)$ at the end of the loop body such that the resulting smaller distributions either enter or exit the loop, respectively. Using the (PAR) rule, our goal becomes to thread-locally verify the code below.

$(\textbf{while}\ (s=0)\ \textbf{do}\ (\textbf{skip}; \langle PWrite_i \rangle\ \textbf{split}(s=0, s\neq 0)\ ))$; $\langle\ SetFlag_i\ ; y_i := s\rangle$,
where $PWrite_i \stackrel{\text{def}}{=} \langle s := i;\ c := c+1; SetFlag_i \rangle \oplus_p \langle\ SetFlag_i\ \rangle$
and $SetFlag_i \stackrel{\text{def}}{=} \textbf{if}\ (s \neq 0)\ \textbf{then}\ d_i := 1\ \textbf{else skip}$

We define the invariant I below, which says that either $s = 0$ (and thus $c = 0$ and each thread has chance to update s), or $s \neq 0$ (and thus $c > 0$) and the probability of $c = 1$ has a lower bound.

$$I \stackrel{\text{def}}{=} I_0 \vee I_1,\ \text{where}\ I_0 \stackrel{\text{def}}{=} \lceil s = 0 \wedge c = 0 \wedge \forall i.\, d_i = 0 \rceil,\ I_1 \stackrel{\text{def}}{=} \lceil s \neq 0 \wedge c > 0 \rceil \wedge \mathsf{PBound},$$
$$\text{and}\ \mathsf{PBound} \stackrel{\text{def}}{=} \exists K \leq n.\, \lceil \textstyle\sum_{i=1}^{n} d_i = K \rceil \wedge \mathbf{Pr}(c = 1) \geq (1-p)^{K-1}$$

We give the detailed proofs in the TR [13]. The logic presented in the paper requires us to split in each round of the while-loop. This technique is sufficient to prove conciliator and *Dice*. However, for more advanced examples such as group election and multiplayer level-up game (in the TR), their loops require split in the first few rounds only. Thus, we extend the logic with a new while rule for while loops and a new sequential composition rule for sequential statements. With the two new rules, we can prove the two advanced examples. The full logic and the proofs of the advanced examples can be found in the TR.

7 Related Work and Discussions

McIver et al. [18] develop the probabilistic rely-guarantee calculus, which, to our knowledge, is the first program logic for concurrent randomized programs. Their semantics assume arbitrary schedules, i.e. the strong adversary (SA) model, and their reasoning rules use probabilistic rely/guarantee conditions. Their logic does not apply to the algorithms of conciliator and group election verified in our work, whose correctness assumes weaker adversary models. Besides, we encode probabilistic properties in the invariant and use only non-probabilistic rely-guarantee conditions, which enable simple stability proofs.

Tassarotti and Harper [20] extend the concurrent program logic Iris [16] with probabilistic relational reasoning, to establish refinements between concurrent randomized programs and monadic models. They also give rules for reasoning about probabilistic properties on monadic models. On the one hand, their program semantics assumes the SA model. On the other hand, their logic soundness only holds for schedules under which the program is guaranteed to certainly terminate (i.e. terminate in a finite number of steps). As a result, they cannot verify the examples in our work.

Fesefeldt et al. [14] propose a concurrent quantitative separation logic for reasoning about lower-bound probabilities of realizing a postcondition of a concurrent randomized program in the SA model. Like us, they require program executions to preserve invariants on shared states. But their invariants are limited to *qualitative* expectations, which map states to either 0 or 1, so they cannot specify probabilistic distributions as ours can. Moreover, they can only verify lower bounds of probabilities, while we can verify exact probabilities and expectations.

For the part of *sequential* reasoning, our rules mostly follow Barthe et al. [6]. Our **lclosed** condition (see the (PAR) rule in Fig. 15) is similar to their "t-closed" condition, both introduced for supporting almost surely terminating programs. Our assertion language for invariants and pre/post-conditions is similar to theirs too, where an assertion is a predicate over state distributions. They provide a (SPLIT) rule which is very different from our split mechanism. Using their (SPLIT) rule, one can logically split the initial distribution into two parts, reason about the execution of the same code on the two parts separately, and mix the two final distributions back. Our (SQ-OPLUS) rule for sequential reasoning in the TR [13], is almost the same as their (SPLIT) rule. It is interesting to extend our assertion language with separating conjunctions, to specify spatial disjointness of state distributions and probabilistic independence (following [7]). There are also (sequential) program logics (e.g. [9,8,1]) where assertions denote functions from program states to probabilities or expected values.

Bertrand et al. [10,11] apply model checking techniques for verifying randomized algorithms in weak adversary models. However, Bertrand et al.'s approach does not apply to the algorithms we have verified. Their work focuses on the class of algorithms with some form of "symmetry" regarding the local control flow. Such an algorithm must execute "symmetric" code for different outcomes of a coin flip. But none of the algorithms verified here satisfies this property. Instead they all have probabilistic branch statements that take different numbers of steps, which is the main challenge to our logic design. We conjecture that our split idea may still be helpful when developing automata-based approaches to verify these algorithms.

Verification overhead and scalability. One may be concerned about the verification overhead caused by adding auxiliary variables and auxiliary code, and the scalability of our logic to large algorithms. In our proofs, auxiliary variables and code are introduced to capture the key intuition of the probabilistic properties that we care about, so they are usually highly related to the random variables and the probabilistic operations (coin flips) in the original algorithms. As a result, the overhead of the auxiliary variables and code is usually proportional to the number of random variables and probabilistic operations rather than the number of lines of code. For large-scale randomized algorithms, the number of probabilistic operations may not be that large, thus the proof overhead of adding auxiliary variables and splits statements should be acceptable.

In our current setting, the auxiliary variables and split statements are added manually during the verification process, which requires a good understanding of the algorithm, i.e., how the algorithm works and why it is correct. We leave it as future work to support automated code instrumentation and verification.

Acknowledgments. We thank anonymous referees for their suggestions and comments on earlier versions of this paper. This work is supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62232015.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Aguirre, A., Barthe, G., Hsu, J., Kaminski, B.L., Katoen, J.P., Matheja, C.: A pre-expectation calculus for probabilistic sensitivity. Proc. ACM Program. Lang. **5**(POPL) (jan 2021). https://doi.org/10.1145/3434333, https://doi.org/10.1145/3434333
2. Alistarh, D., Aspnes, J.: Sub-logarithmic test-and-set against a weak adversary. In: Proceedings of the 25th International Conference on Distributed Computing. pp. 97–109. DISC'11, Springer-Verlag, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24100-0_7
3. Aspnes, J.: Randomized protocols for asynchronous consensus. Distributed Comput. **16**(2-3), 165–175 (2003). https://doi.org/10.1007/s00446-002-0081-5, https://doi.org/10.1007/s00446-002-0081-5
4. Aspnes, J.: Notes on randomized algorithms (2023), https://www.cs.yale.edu/homes/aspnes/classes/469/notes.pdf
5. Aspnes, J.: Notes on theory of distributed systems (2023), https://www.cs.yale.edu/homes/aspnes/classes/465/notes.pdf
6. Barthe, G., Espitau, T., Gaboardi, M., Grégoire, B., Hsu, J., Strub, P.: An assertion-based program logic for probabilistic programs. In: Proceedings of the 27th European Symposium on Programming (ESOP 2018). pp. 117–144. Springer (2018). https://doi.org/10.1007/978-3-319-89884-1_5, https://doi.org/10.1007/978-3-319-89884-1_5
7. Barthe, G., Hsu, J., Liao, K.: A probabilistic separation logic. Proc. ACM Program. Lang. **4**(POPL), 55:1–55:30 (2020)
8. Batz, K., Kaminski, B.L., Katoen, J.P., Matheja, C.: Relatively complete verification of probabilistic programs: An expressive language for expectation-based reasoning. Proc. ACM Program. Lang. **5**(POPL) (jan 2021). https://doi.org/10.1145/3434320, https://doi.org/10.1145/3434320
9. Batz, K., Kaminski, B.L., Katoen, J.P., Matheja, C., Noll, T.: Quantitative separation logic: A logic for reasoning about probabilistic pointer programs. Proc. ACM Program. Lang. **3**(POPL) (jan 2019). https://doi.org/10.1145/3290347, https://doi.org/10.1145/3290347
10. Bertrand, N., Konnov, I., Lazic, M., Widder, J.: Verification of randomized consensus algorithms under round-rigid adversaries. In: Fokkink, W.J., van Glabbeek, R. (eds.) Proceedings of the 30th International Conference on Concurrency Theory (CONCUR 2019). LIPIcs, vol. 140, pp. 33:1–33:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.CONCUR.2019.33, https://doi.org/10.4230/LIPIcs.CONCUR.2019.33
11. Bertrand, N., Lazic, M., Widder, J.: A reduction theorem for randomized distributed algorithms under weak adversaries. In: Henglein, F., Shoham, S., Vizel, Y. (eds.) Proceedings of the 22nd International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2021). Lecture Notes in Computer Science, vol. 12597, pp. 219–239. Springer (2021). https://doi.org/10.1007/978-3-030-67067-2_11, https://doi.org/10.1007/978-3-030-67067-2_11
12. Chor, B., Israeli, A., Li, M.: Wait-free consensus using asynchronous hardware. SIAM Journal on Computing **23**(4), 701–712 (1994). https://doi.org/10.1137/S0097539790192635, https://doi.org/10.1137/S0097539790192635
13. Fan, W., Liang, H., Feng, X., Jiang, H.: A program logic for concurrent randomized programs in the oblivious adversary model. Tech. rep. (2025), https://plax-lab.github.io/publications/randoa/randoa-tr.pdf

14. Fesefeldt, I., Katoen, J., Noll, T.: Towards concurrent quantitative separation logic. In: Proceedings of 33rd International Conference on Concurrency Theory (CONCUR 2022). pp. 25:1–25:24 (2022). https://doi.org/10.4230/LIPIcs.CONCUR.2022.25, https://doi.org/10.4230/LIPIcs.CONCUR.2022.25
15. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4), 596–619 (oct 1983). https://doi.org/10.1145/69575.69577, https://doi.org/10.1145/69575.69577
16. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 637–650. POPL '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2676726.2676980, https://doi.org/10.1145/2676726.2676980
17. McIver, A., Morgan, C.: Abstraction, Refinement and Proof for Probabilistic Systems. Monographs in Computer Science, Springer (2005). https://doi.org/10.1007/b138392, https://doi.org/10.1007/b138392
18. McIver, A., Rabehaja, T.M., Struth, G.: Probabilistic rely-guarantee calculus. Theor. Comput. Sci. **655**, 120–134 (2016). https://doi.org/10.1016/j.tcs.2016.01.016, https://doi.org/10.1016/j.tcs.2016.01.016
19. Rand, R., Zdancewic, S.: VPHL: A verified partial-correctness logic for probabilistic programs. In: Ghica, D.R. (ed.) Proceedings of the 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS 2015). Electronic Notes in Theoretical Computer Science, vol. 319, pp. 351–367. Elsevier (2015). https://doi.org/10.1016/j.entcs.2015.12.021, https://doi.org/10.1016/j.entcs.2015.12.021
20. Tassarotti, J., Harper, R.: A separation logic for concurrent randomized programs. Proc. ACM Program. Lang. **3**(POPL) (jan 2019). https://doi.org/10.1145/3290377, https://doi.org/10.1145/3290377