Probabilistic Total Store Ordering

Parosh Aziz Abdulla\textsuperscript{1}, Mohamed Faouzi Atig\textsuperscript{1}, Raj Aryan Agarwal\textsuperscript{2}, Adwait Godbole\textsuperscript{3}, and Krishna S.\textsuperscript{2}

\textsuperscript{1} Uppsala University, Sweden  \textsuperscript{2} IIT Bombay, India  \textsuperscript{3} University of California Berkeley, USA adwait@berkeley.edu

Abstract. We present Probabilistic Total Store Ordering (PTSO) – a probabilistic extension of the classical TSO semantics. For a given (finite-state) program, the operational semantics of PTSO induces an infinite-state Markov chain. We resolve the inherent non-determinism due to process schedulings and memory updates according to given probability distributions. We provide a comprehensive set of results showing the decidability of several properties for PTSO, namely (i) Almost-Sure (Repeated) Reachability: whether a run, starting from a given initial configuration, almost surely visits (resp. almost surely repeatedly visits) a given set of target configurations. (ii) Almost-Never (Repeated) Reachability: whether a run from the initial configuration, almost never visits (resp. almost never repeatedly visits) the target. (iii) Approximate Quantitative (Repeated) Reachability: to approximate, up to an arbitrary degree of precision, the measure of runs that start from the initial configuration and (repeatedly) visit the target. (iv) Expected Average Cost: to approximate, up to an arbitrary degree of precision, the expected average cost of a run from the initial configuration to the target. We derive our results through a nontrivial combination of results from the classical theory of (infinite-state) Markov chains, the theories of decisive and eager Markov chains, specific techniques from combinatorics, as well as, decidability and complexity results for the classical (non-probabilistic) TSO semantics. As far as we know, this is the first work that considers probabilistic verification of programs running on weak memory models.

1 Introduction

The classical Sequential Consistency (SC) semantics [1] has been a fundamental assumption in concurrent programming. SC guarantees that process operations are atomic. A write operation, performed by a given process, is immediately visible to all the other processes. However, designers of modern computer systems, in their quest of increased system efficiency, often sacrifice the SC guarantee. Instead, the processes communicate asynchronously, allowing a delay in the propagation of write operations. Due to the propagation delay, written values can become available to processes at different time points, and in an order that may be different from the order in which they are generated. This asynchronous behavior gives rise to new semantics, collectively referred to as weak memory...
models [2]. In the presence of weak memory models, programs exhibit new, and often unexpected, behaviors, bringing about complex challenges in the design and analysis of concurrent systems. Even text-book programs may behave erroneously. The classical Dekker mutual exclusion protocol is a case in point. The ubiquity of weak memory models has led to an extensive research effort for the testing and verification of concurrent programs running under such semantics.

Existing works on the verification of programs running on weak memory models, consider safety properties such as state reachability, assertion violation, and robustness. While safety properties are fundamental, we need also to prove liveness properties, i.e., to show that the program indeed makes progress. This is, of course, true already in the case of SC. A program, such as a mutual exclusion protocol, needs to guarantee that each process will eventually reach its critical section. The satisfiability of liveness properties is often dependent on the type of fairness conditions on process executions that are provided by the underlying platform [3,4]. The reason is the presence of concurrency non-determinism, i.e., the inherent non-determinism in program behavior due to the different possible ways in which the scheduler can interleave the processes. The scheduler may always neglect a given process, which means that the process will never make progress (e.g., never reaches its critical section). Therefore, we need the scheduler to follow a fair selection policy that allows each process to advance in its execution. The situation is even more complicated in the case of weak memory models, since we also need to deal with a second source of non-determinism, besides concurrency non-determinism, namely (data) propagation non-determinism. Since write operations are propagated asynchronously, there is in general no way to predict if, when, and in which order, write operations become visible to the processes.

In this paper we present a framework for the verification of liveness properties for concurrent programs running under the classical Total Store Ordering (TSO) semantics [5]. The TSO model puts an unbounded store (write) buffer between each process and the main memory. The buffer carries pending write operations that have been performed by the process. These operations are propagated from the buffer to the shared memory in a FIFO manner. When a process performs a write operation, it appends the operation as a message to its buffer. When a process reads a variable, it searches its buffer for a pending write operation on that variable. If such operations exist then it reads from the most recent one. If no such operation exists, it fetches the value of the variable from the main memory. The TSO propagation mechanism is a typical example of how propagation non-determinism arises: the write operations are propagated to the shared memory non-deterministically, and a process sees the other processes’ write operations only when the latter are available in the memory. Therefore, having a scheduler that fairly selects the processes is not sufficient. We also need to ensure that the write operations propagate to the processes sufficiently often.

Traditional fairness conditions such as strong or weak fairness [3,4,6] cannot capture propagation policies adequately since they irrationally allow slow propagation, i.e., they allow write operations to propagate at a lower rate than
the rate by which they are issued. For instance, strong fairness guarantees that messages are transferred infinitely often from the buffers to the memory. Still, it does not constrain the relative frequency of write and update operations, and hence it does not prevent the buffer contents from growing unboundedly. In such a scenario, more and more un-propagated messages may be clustered inside the buffers, and a given process may, from some point on, be confined only to read its own writes, since it will not see the memory updates by the other processes. Accordingly, verifying liveness properties subject to strong fairness may wrongly deem the system to be incorrect: even if a process is selected infinitely often by the scheduler and write operations are propagated infinitely often to the memory, a given process may incorrectly be judged not to make progress due to slow propagation.

While slow propagation can arise theoretically under the above mentioned fairness conditions, it is almost never observed in practice. Existing platforms implement different policies, such as invalidation or write-back policies, to flush the buffers at regular intervals [7,8]. This prevents the buffer sizes from growing beyond certain sizes, and implicitly ensure propagation fairness. In fact, this is true to the degree that non-SC behaviors are (relatively) rarely observed on TSO platforms [9,10].

In this paper, we perform verification of liveness properties for concurrent programs under TSO using probabilistic fairness [11]. As far as we know, this is the first work that considers probabilistic verification of programs running on weak memory models. In our model, both process scheduling and message propagation are carried out according to given probability distributions. We assign a weight (a natural number) to each process. We resolve concurrency non-determinism probabilistically by letting the scheduler select the next process to execute with a probability that reflects the weight of the process compared to the weights of the other processes that are enabled in the same configuration. After each process step, we allow an update step, in which the buffers transfer parts of their contents to the memory. We make the probability distribution equal among all possible update operations in the given configuration. As we will see later in the paper, defining the model in this way implies that we assign low probabilities to program runs that unboundedly increase the number of messages inside the buffers. Accordingly, our model is more faithful to real program behavior compared to models induced by non-probabilistic fairness conditions.

We perform a comprehensive analysis of the decidability of verifying liveness properties for concurrent programs running under the TSO semantics, subject to probabilistic fairness. In fact, verifying programs running on the TSO memory model, even with respect to safety properties, poses a difficult challenge. The unboundedness of the buffers implies that the state space of the system is infinite, even in the case where the input program is finite-state [12,13]. Similarly, the operational semantics of our model gives rise to Markov chains with infinite state spaces. Furthermore, in general, liveness properties give rise to more difficult problems than safety properties, since the former are interpreted over

---

4 Our framework allows several other types of probability distributions (see Sec. 9.)
infinite program executions while the latter are interpreted over finite executions. Our results rely on nontrivial combinations of results from the classical theory of (infinite-state) Markov chains \cite{14,15}, the theories of decisive and eager Markov chains \cite{16,17}, specific techniques from combinatorics \cite{18}, as well as, decidability and complexity results for the classical (non-probabilistic) TSO semantics \cite{19,13}. Concretely, we show the decidability of the following problems, each of which is defined by giving an initial configuration $\gamma_{\text{init}}$ and a set $\text{Target}$ of process target states.

**Qualitative Analysis** (Sec. 6). In qualitative reasoning, we are interested in knowing whether the given property is satisfied with probability 1 (almost surely satisfied), or with probability 0 (almost never satisfied). We show that the satisfiability of these properties can be reduced to similar problems on the underlying (non-probabilistic) transition systems for classical TSO. The actual probabilities appearing in the induced Markov chains then are inconsequential and only their non-zeroness matters. This is useful whenever the probabilities have not been measured exactly, or the portion of the system giving rise to probabilistic behavior has not been designed yet. We consider the following different flavors of qualitative analysis: **Almost-Sure (Repeated) Reachability**\(^5\): whether a run of the system from $\gamma_{\text{init}}$ will almost surely visit (resp. repeatedly visit) $\text{Target}$; **Almost-Never (Repeated) Reachability**: whether a run of the system from $\gamma_{\text{init}}$ will almost never visit (resp. repeatedly visit) $\text{Target}$. Furthermore, we show that all these problems have non-primitive-recursive complexities.

**Quantitative Analysis** (Sec. 7). The task is to estimate to an arbitrary degree of precision the probability by which a run from $\gamma_{\text{init}}$ (repeatedly) visits $\text{Target}$, rather than only checking whether the probability is equal to one or zero.

**Expected Average Cost** (Sec. 8). We study the expected cost for runs that start from $\gamma_{\text{init}}$ until they reach $\text{Target}$. To that end, we extend our model by providing a cost function that assigns a fixed cost to each instruction in the language. Calculating expected costs of runs has many potential applications. For instance, one might be interested in the mean-time of reaching a target, i.e., the average number of steps before reaching the target \cite{20}. In the context of weak memory models, in general, and TSO in particular, one can perform a more refined analysis by also taking into account the fact that specific instructions, e.g., memory fences, have higher costs \cite{21}. Incorporating instruction costs in the model makes average cost analysis reflect more faithfully the efficiency of the program compared to an instruction count based metric. There have been several approaches towards optimizing fence implementations in hardware \cite{22,23,24} which exploit the fact that non-SC behaviours are rare even in unfenced code. A quantitative analysis of the prevalence of behaviours and cost of executing instructions can help determine the efficacy of such implementations.

---

\(^5\) While repeated reachability is a liveness property, plain reachability in the non-probabilistic case is a safety property. However, in the presence of probabilities, plain reachability measures the probability of convergence towards a target state, and hence it can be considered a form of liveness property. In any case, this is a matter of definition and has no bearing on the rest of the paper.
The supplementary material [25] contains detailed proofs of all the lemmas and theorems.

**Related Work** Only recently there has been an increased interest in the formulation and verification of liveness properties for weak memory models. In [26], they factor the system into a process and memory subsystems and define notions of fairness for either. This is reminiscent of our approach, where we consider probabilistic policies for process scheduling and memory update. Their model on the other hand is non-probabilistic and they have weaker fairness guarantees, which we describe in more detail in Sec. 5.1. The liveness verification problem for TSO has been considered in [27], where they show undecidability for various liveness properties. However, once again work with non-probabilistic notions of fairness. We show in this paper, that with stronger (probabilistic) fairness, reachability and repeated reachability problems become decidable.

In [12], they show the undecidability of the repeated reachability problem, *without* fairness conditions, for finite-state programs running under the TSO semantics. In contrast, we show that checking repeated reachability qualitatively is decidable (Sec. 6.2), and that we can even compute the measure of runs satisfying the property with arbitrary precision (Sec. 7.2).

There has been a huge amount of work on the verification of finite-state Markov chains (see, e.g., [20,28]). Since the buffers in TSO are unbounded, we however, get an infinite-state Markov chain. There is also a substantial literature on the verification of infinite-state Markov chains, where specialized techniques are developed for particular classes of systems. Several works have considered probabilistic push-down automata and probabilistic recursive machines [29,30,31]. However, these techniques don’t apply in our case since push-down automata cannot encode the FIFO store-buffer data-structure.

Works such as [32,16,33,34] develop algorithmic and complexity results for checking termination and reachability for systems such as probabilistic VASS, probabilistic Petri nets, probabilistic multi-counter systems. Again, these models are different from ours and cannot encode FIFO queues.

The works closest to ours are those on probabilistic lossy channel systems [16,17]. These works also rely on the frameworks of decisive and eager Markov chains. However, lossy channel systems and TSO are fundamentally different, and the manner in which we instantiate the frameworks of decisive/eager Markov chains differs. The decidability of verification for probabilistic extensions of lossy channels is sensitive to the definition of the message losses. In the case of lossy channel systems, if messages are only allowed to be lost at one end of the channel (a model that is close to our notion of message updates), then all non-trivial verification problems become undecidable for probabilistic lossy channel systems [35]. Therefore, although there is a reduction from TSO to lossy channel systems in the case of non-probabilistic models [12], we know of no such reduction between the corresponding probabilistic models.

Finally, the concept of decisiveness has been extended to more general models such as generalized semi-Markov processes, stochastic timed automata [36], and lossy channel-based stochastic games [37].
2 Preliminaries

In this section, we introduce notation, recall basics of transition systems, Temporal logic and Markov chains.

Basic Notation The size of a set $A$ is denoted by $|A|$. We use $A^*$ and $A^\omega$ to denote the set of finite resp. infinite words over (a possibly infinite set) $A$, and let $\epsilon$ be the empty word. For $w \in A^*$, $|w|$ denotes the length of $w$ ($|w| = \infty$ if $w$ is infinite). For $i : 1 \leq i \leq |w|$, we use $w[i]$ to denote the $i^{th}$ element of $w$. We define $\text{head}(w) := w[1]$ and $\text{tail}(w) := w[2] \cdots w[|w|]$. We use $a \in w$ to denote that $w[i] = a$ for some $i : 1 \leq i \leq |w|$. For words $w_1 \in A^*$ and $w_2 \in (A^* \cup A^\omega)$, we use $w_1 \cdot w_2$ to denote their concatenation. For $k \in \mathbb{N}$, we define $A^k := \{ w \in A^* \mid |w| = k \}$, i.e., it is the set of words over $A$ of length $k$.

Transition Systems A transition system is a pair $\langle \Gamma, \rightarrow \rangle$ where $\Gamma$ is a (potentially) infinite set of configurations, and $\rightarrow \subseteq \Gamma \times \Gamma$ is the transition relation. We write $\gamma \rightarrow \gamma'$ to denote that $\langle \gamma, \gamma' \rangle \in \rightarrow$, and use $\gamma \trans$ to be the reflexive transitive closure of $\rightarrow$. For $k \in \mathbb{N}$, we write $\gamma \trans^k \gamma'$ to denote that there is a sequence $\gamma_0 \rightarrow \gamma_1 \rightarrow \cdots \rightarrow \gamma_k$ where $\gamma_0 = \gamma$ and $\gamma_k = \gamma'$, i.e., there is a sequence of $k$ transition steps leading from $\gamma$ to $\gamma'$. For $\gamma \in \{ <, \leq, =, \geq, > \}$, we write $\gamma \trans^k \gamma'$ to denote that $\gamma \trans^m \gamma'$ for some $m : 0 \leq m \sim k$.

Temporal Logic A run $\rho$ of transition system $T = \langle \Gamma, \rightarrow \rangle$ is an infinite word $\gamma_0 \gamma_1 \ldots$ of configurations such that $\gamma_i \rightarrow \gamma_{i+1}$ for $i \geq 0$. We use $\rho[i]$ to denote $\gamma_i$. We say that $\rho$ is a $\gamma$-run if $\rho[0] = \gamma$. We use Runs $\gamma$ to denote the set of $\gamma$-runs. A path $\pi$ is a finite prefix of a run, and a $\gamma$-path is a finite prefix of a $\gamma$-run. We use the standard notation $\gamma \models T \phi$ to represent that $\gamma$ satisfies the $CTL^*$ state formula $\phi$ and $\rho \models T \phi$ to mean that $\rho$ satisfies the path formula $\phi$. We refer the reader to [38] for details of CTL.

For $\gamma \in \Gamma$ and $G \subseteq \Gamma$, we say that $G$ is reachable from $\gamma$, denoted $\gamma \models T \exists \gamma \in G$, if there is a $\gamma$-run $\rho$ such that $\rho[i] \in G$ for some $i$. For $k \in \mathbb{N}$, $\gamma \in \Gamma$, and $G \subseteq \Gamma$, $\rho \models T \gamma^k G$ says that $\rho$ reaches $G$ first at the $k^{th}$ step. For $\sim \in \{ <, \leq, =, \geq, > \}$, $\rho \models T \gamma^k G$ says that $\rho \models T \gamma^m G$ holds for some $m : 0 \leq m \sim k$. The statement $\rho \models T \gamma^k G$ says that $\rho$ visits $G$ at the $k^{th}$ step (but possibly earlier).

Markov Chains A Markov chain $C$ is a pair $\langle \Gamma, \mathcal{M} \rangle$ where $\Gamma$ is a (potentially infinite) set of configurations, and $\mathcal{M} : \Gamma \times \Gamma \rightarrow [0, 1]$ is a transition probability matrix over $\Gamma$, called the probability matrix of $C$, i.e. $\mathcal{M}$ satisfies: $\forall a \in A. \sum b \in A \mathcal{M}(a, b) = 1$. A Markov chain $C = \langle \Gamma, \mathcal{M} \rangle$ induces an underlying transition system, denoted $C^\downarrow$. We define $C^\downarrow := \langle \Gamma, \rightarrow \rangle$, where $\rightarrow := \{ \langle \gamma, \gamma' \rangle \mid \mathcal{M}(\gamma, \gamma') > 0 \}$. The underlying transition system has the same configuration set, with transitions between configurations that have non-zero transition probability under $C$. This allows us to lift the temporal logic concepts defined above to Markov chains.

---

6 We term infinite sequences as runs and finite sequences as paths. However, traditionally, $CTL^*$ refers to properties of infinite-sequences (our runs) as path-formulae.
**Probability Measures** Consider a Markov chain $C = \langle \Gamma, M \rangle$. The probability of taking path $\pi$ is the product of single step probabilities along $\pi$:

$$\text{Prob}_C (\pi) := \prod_{i=0, \ldots, |\pi|-1} M(\pi[i], \pi[i+1])$$

For a configuration $\gamma$, we adopt the usual probability space on $\gamma$-runs with the $\sigma$-algebra over cylindrical sets starting from $\gamma$ (see [39, 20] for details). For path formula $\phi$, we define $\text{Prob}_C (\gamma \models \phi) = \text{Prob}_C (\{ \rho \in \text{Runs} (\gamma) \mid \rho \models_C \phi \})$ (which is measurable by [40]), e.g. given a set $F \subseteq G$, $\text{Prob}_C (\gamma \models \Diamond F)$ is the measure of $\gamma$-runs which reach $F$. If $\text{Prob}_C (\gamma \models \phi) = 1$ the we say that almost all $\gamma$-runs of $C$ satisfy $\phi$. Following the literature, we say that $\gamma \models_C \phi$ holds almost surely (almost certainly), or that $\phi$ holds almost surely from $\gamma$.

### 3 Concurrent Programs

A (concurrent) program consists of a set of *processes* that run in parallel and communicate through a set of *shared variables*. The operation of the program is controlled by a central *scheduler* that selects the processes to execute one after the other. We assume a finite set $\text{Procs}$ of processes that share a set $\mathcal{X}$ of variables. Fig. 1 gives the grammar for a small but general assembly-like language that we use for defining the syntax of concurrent programs. A program instance, $P$ is described by a set of shared variables, $\text{var}^\ast$, followed by the codes of the processes, $(\text{proc reg}^\ast \text{instr}^\ast)^\ast$. Each process $p \in \text{Procs}$ has a finite set $\text{Regs}_p$ of (local) registers. We assume that the sets of registers of the different processes are disjoint, and define $\text{Regs}_P := \bigcup_{p \in \text{Procs}} \text{Regs}_p$. Each process declares its set of registers, $\text{reg}^\ast$, followed by a sequence of instructions. We assume that the data domain of $\mathcal{X}$ and $\text{Regs}_P$ is a finite set $\mathcal{V}$, with a special element $0 \in \mathcal{V}$.

### Instructions

An instruction $i$ is of the form $l : s$ where $l$ is a unique (across processes) label and $s$ is a statement. Labels represent program counters of processes and indicate the instruction that the process executes the next time it is scheduled. A *read/write* statement either writes the value of a register to a shared variable, reads the value of a shared variable into a register, or updates the value of a register by evaluating an expression. We assume a set $\text{expr}$ of expressions over constants and registers, but not referring to the shared variables. The $\text{CAS}$ statement is the standard compare-and-swap operation, and $\text{if}$-statements have their usual interpretations. Iterative constructs such as $\text{while}$ and $\text{for}$, as well as $\text{goto}$-statements, can be encoded with branching $\text{if}$-statements as usual.
The `fence` statement, that flushes the contents of the buffer of the process, can be simulated using the `CAS` statement. The statement `term` will cause the process to terminate its execution. Sometimes, we will refer to an instruction by its statement, e.g. the instruction `r := x`, (where `r` is a register and `x` is a shared variable) a `read` instruction, similarly for a `write` instruction, etc. Semantics of these instructions are explained through a set of inference rules in Sec. 4.

**Labels** We define `Lbl_p` to be the set of labels that occur in the code of the process `p`, and define `Lbl_P := ∪_{p ∈ Procs} Lbl_p`. We assume that `term` has the label `term_p`. We define `Instr_p` to be the set of instructions occurring in `p`, and define `Instr_P := ∪_{p ∈ Procs} Instr_p`. For instruction `i` of the form `l : s` we define `λ(i) := l` and `stmt(i) := s`. Abusing notation, we also define `stmt(l) := s`. For a process `p ∈ Procs` instruction `i ∈ Instr_p`, with `stmt(i) ≠ term`, we define `next(i)` to be the (unique) instruction next to `i` in the code of `p`. For an instruction `l_1 : (if a then l_2)`, we assume, without loss of generality\(^7\), that `l_1 ≠ l_2`.

**Scheduler** The scheduler selects the process from `Procs` to run next. The operational model for classical TSO \([41]\) uses a non-deterministic scheduler. We adopt a scheduler that selects the next process probabilistically. The scheduler policy is defined by a function `Sched : Sched(p) ∈ N` denotes the scheduling weight assigned to to the process `p`. If `p` is `enabled` (i.e. the process can execute the next instruction, formally defined in Sec. 4) then `p` is scheduled at the next step with a probability that is proportional to `Sched(p)`.

### 4 Operational Semantics

The operational model for classical TSO \([41]\) describes the semantics as a transition system. We also take an operational approach. However, we differ in a fundamental aspect: classical TSO models choice between transitions as non-deterministic choice. We on the other hand, model this as probabilistic choice, to get a system called as Probabilistic TSO (PTSO for short). Adding probabilities induces a Markov chain, which governs the behaviours of PTSO.

A program is described by a pair: the set of processes, `Procs` and the scheduler policy `Sched`. In this section, we fix such a program `P = ⟨Procs, Sched⟩`. We develop the operational semantics of `P` under PTSO as an infinite-state Markov chain \([P]_{MC} := ⟨Γ_P, M_P⟩\). We begin by defining the set of configurations `Γ_P` (Sec. 4.1). Then we describe the behavior of `P` under classical TSO using a transition system \([P]_{TS}\) (Sec. 4.2); Finally, we extend the transition system to a Markov chain \([P]_{MC}\) by giving probability distributions that define govern process scheduling, and memory updates.

\(^7\) We make the restriction for technical convenience. The case where `l_1 = l_2` do not introduce conceptual difficulties. However, it simplifies the presentation by eliminating some corner cases when we define probability measures (Sec. 5) and when we introduce our cost model (Sec. 8).
4.1 Configurations

The central feature of TSO is the store buffer: a FIFO buffer in which pending write operations are queued as messages. The semantics equips each process \( p \in \text{Procs} \) with an unbounded buffer, here called the \( p \)-buffer, that carries pending write operations issued by \( p \), but that have yet not reached the shared memory.

A configuration, \( (\lambda, \mathcal{R}, \mathcal{B}, \mathcal{M}) \), describes four attributes: a labeling state \( (\lambda) \), a register state \( (\mathcal{R}) \), a buffer state \( (\mathcal{B}) \), and a memory state \( (\mathcal{M}) \). We use \( \Gamma_P \) to denote the set of configurations of \( P \).

A labeling state is a function \( \lambda : \text{Procs} \to \text{Lbl}_P \) that defines, for \( p \in \text{Procs} \), the label \( \lambda(p) \in \text{Lbl}_p \) of the next instruction to be executed by \( p \).

A register state is a function \( \mathcal{R} : \text{Regs} \to \mathcal{V} \) that maps each register \( a \in \text{Regs}_P \), to its current value \( \mathcal{R}(a) \in \mathcal{V} \). For an expression \( e \), we use \( \mathcal{R}(e) \) to denote the evaluation of \( e \) against the register state \( \mathcal{R} \).

A single-buffer state \( w \) is a word in \( (\mathcal{X} \times \mathcal{V})^\ast \), describing the content of the \( p \)-buffer for some process \( p \in \text{Procs} \). The buffer contains a sequence of pending write messages, i.e., pairs of form \( \langle x, v \rangle \) representing a write to \( x \), with value \( v \).

A buffer state is a function \( \mathcal{B} : \text{Procs} \to (\mathcal{X} \times \mathcal{V})^\ast \) that defines, for each process \( p \in \text{Procs} \), a single-buffer state describing the content of the \( p \)-buffer.

A memory state is a function \( \mathcal{M} : \mathcal{X} \to \mathcal{V} \) that assigns to each variable \( x \in \mathcal{X} \) its current value \( \mathcal{M}(x) \in \mathcal{V} \) in the shared memory.

\[
\begin{array}{|c|c|c|c|}
\hline
\text{write} & \text{read} & \text{expr} \\
\hline
\text{stmt (λ(p)) = } (x := a) & \text{FetchVal (x) (B(p)) (M(x)) = v} & \text{stmt (λ(p)) = (a := e)} \\
B' = B[p ← (x, R(a) : B'[p])] & \lambda' = \lambda[p ← \text{next (λ(p))}] & R' = R[a ← v] \\
\langle \lambda, R, B, M \rangle \xrightarrow{\text{prog}} (X', R', B', M') & \langle \lambda, R, B', M' \rangle \xrightarrow{\text{prog}} (X', R', B, M) \\
\langle \lambda, R, B, M \rangle \xrightarrow{\text{prog}} (X', R', B', M') & \langle \lambda, R, B', M' \rangle \xrightarrow{\text{prog}} (X', R', B, M) \\
\hline
\text{CAS-true} & \text{CAS-false} & \text{proc} & \text{disabled} \\
\text{stmt (λ(p)) = (b := CAS (x, a, a2))} & \text{stmt (λ(p)) = (b := CAS (x, a, a2))} & \text{proc (λ, R, B, M)} \xrightarrow{\text{prog}} (X', R', B', M') & \gamma \text{ is disabled} \\
R' = R[b ← false] & M(x) = R(a) & \langle \lambda, R, B, M \rangle \xrightarrow{\text{prog}} (X', R', B', M') & \gamma \xrightarrow{\text{prog}} \gamma' \\
\lambda' = \lambda[p ← \text{next (λ(p))}] & M(x) = M[x ← R(a2)] & \langle \lambda, R', B', M' \rangle \xrightarrow{\text{prog}} (X', R', B', M') & \langle \lambda, R, B, M \rangle \xrightarrow{\text{prog}} (X', R', B', M') \\
\langle \lambda, R, B, M \rangle \xrightarrow{\text{prog}} (X', R', B', M') & \langle \lambda, R, B', M' \rangle \xrightarrow{\text{prog}} (X', R', B, M) & \langle \lambda, R', B', M' \rangle \xrightarrow{\text{prog}} (X', R', B', M') & \langle \lambda, R, B, M \rangle \xrightarrow{\text{prog}} (X', R', B', M') \\
\hline
\end{array}
\]

Fig. 2. The classical TSO semantics: process transitions (green), update transitions (orange) and overall transition (Full-\text{TOSO})

Consider a configuration \( \gamma = (\lambda, R, B, M) \). We say that \( \gamma \) is plain if \( B(p) = \epsilon \) for all \( p \in \text{Procs} \), i.e., all the buffers in \( \gamma \) are empty. We use \( \Gamma_P^{\text{plain}} \) to denote the set of plain configurations of \( P \). Notice that \( \Gamma_P^{\text{plain}} \subseteq \Gamma_P \) and that \( \Gamma_P^{\text{plain}} \) is...
finite. For a label \(l \in \text{Lbl}_P\), we write \(l \in \gamma\) if \(\lambda(p) = l\) for some \(p \in \text{Procs}\). We define \(\Gamma^l_P := \{\gamma \in \Gamma_P \mid l \in \gamma\}\), i.e., configurations in which \(l\) occurs.

For a configuration \(\gamma = \langle \lambda, \mathcal{R}, B, \mathcal{M} \rangle\) we define the size of \(\gamma\) by \(|\gamma| := \sum_{p \in \text{Procs}} |B(p)|\), i.e., it is the total number of messages in the buffers in \(\gamma\). For \(\sim \in \{<, \leq, =, \geq, >\}\), we define \(\Gamma^\sim_P := \{\gamma \in \Gamma_P \mid |\gamma| \sim \ell\}\), i.e. configurations where the total number of messages, \(m\), relates to \(\ell\) by \(m \sim \ell\).

### 4.2 The Classical TSO Semantics

We recall the classical semantics of TSO, using a transition system \([P]^{TS} = (\Gamma_P, \rightarrow_P)\). We define the transition relation \(\rightarrow_p\) through the set of inference rules in Fig. 2. The relation \(\rightarrow_P\) is the composition of two relations: the relation \(\rightarrow_{\text{proc}}\) describes the processes’ execution steps, and the relation \(\rightarrow_{\text{update}}\) describes memory updates, where pending writes are propagated to the memory.

**Process Transitions** We define the process transition relation \(\rightarrow_{\text{proc}} := \bigcup_{p \in \text{Procs}} \mathcal{P}_p\) as a union of relations each corresponding to one process (the rule proc). The inference rules defining \(\mathcal{P}_p\), for a process \(p \in \text{Procs}\) are depicted in Fig. 2. Each rule corresponds to one step performed by \(p\). After executing an instruction, \(p\) will move on to the next instruction in its code. It executes the latter instruction when again selected by the scheduler.

A write instruction \((x := a)\) assigns the value of the local register \(a\) to the shared variable \(x\). The process appends a write message consisting of \(x\) together with the value \(\mathcal{R}(a)\) of \(a\), to the head of the \(p\)-buffer. A read instruction, \((a := x)\), assigns the value of the shared variable \(x\) to the local register \(a\). The value of \(x\) is either fetched from the \(p\)-buffer (read-own-write), or from the shared memory (read-from-memory). We capture both cases in one inference rule, using the function \text{FetchVal} defined as follows. Let \(w\) be the contents of the \(p\)-buffer. We write \(x \in w\) if \((x,v) \in w\) for some \(v \in \mathcal{V}\), and write \(x \notin w\) otherwise. We define (i) \(\text{FetchVal}(x)(w)(\mathcal{M}) := v\) if \(x \in w\) and \(w = w_1 \cdot (x,v) \cdot w_2\) with \(x \notin w_1\); and (ii) define \(\text{FetchVal}(x)(w)(\mathcal{M}) := \mathcal{M}(x)\) if \(x \notin w\). In case (i), the value of \(x\) is taken from the latest \(x\)-message from the \(p\)-buffer. In case (ii), no \(x\)-messages exist in the \(p\)-buffer, and the value is read from the shared memory.

The instruction \(b := \text{CAS}(x,a_1,a_2)\) checks whether the \(p\)-buffer is empty and the value of the shared variable \(x\) is equal to the value of the register \(a_1\). If yes, we assign atomically the value of the register \(a_2\) to \(x\), and assign the value true to \(b\) (the rule CAS-true). If the value of \(x\) is different from the value of \(a_1\) then we do not change the value of \(x\), but assign the value false to \(b\) (the rule CAS-false). If the \(p\)-buffer is not empty then \(p\) is disabled in the current configuration. We define the set of disabled processes at configuration \(\gamma\):

\[
\text{disab}(\gamma) := \{p \mid (\text{stmt}(p) = \text{term}) \lor ((\text{stmt}(p) = (b := \text{CAS}(x,a_1,a_2))) \land (B(p) \neq \epsilon))\}
\]

In other words, it is the set of processes that are disabled in \(\gamma\) either because they have terminated or because they are about to perform a CAS operation.
and their buffers are not empty. We say that \( p \) is \textit{disabled} in \( \gamma \) if \( p \in \text{disab}(\gamma) \), and that \( \gamma \) is \textit{disabled} if all the processes are disabled in \( \gamma \). If a process (resp. configuration) is not disabled then it is enabled. If \( \gamma \) is disabled, we make a dummy transition that does not change \( \gamma \) (the rule \textit{disabled})\(^8\). Notice that if \( \gamma \xrightarrow{\text{proc}} \gamma' \) then there is unique process \( p \in \text{Procs} \) such that \( \gamma \xrightarrow{p} \gamma' \).

**Update Transitions** Between two process transitions, the system may perform a (possibly empty) sequence of update steps. The rule \textit{empty-update} describes an empty update step. Each \textit{single-update} step pops one write message at the end of the \( p \)-buffer for some process \( p \) and uses it to update the memory. The \textit{update} rule captures the effect of a sequence of such \textit{single-update} steps. We define the update transition relation \( \xrightarrow{\text{update}} := \bigcup_{\alpha \in \text{Procs}} \alpha \xrightarrow{\text{update}} \) as a union of relations each corresponding to a given sequence of update steps. The word \( \alpha \) gives the sequence of processes that perform the updates. The net effect is that the system (i) pops a sequence of (possibly empty) suffixes from the buffer of each process, (ii) shuffles these into one sequence, and (iii) uses the resulting sequence to update the memory. Notice that each selection of possible suffixes in step (i) may result in several different sequences due to multiple interleavings in step (ii). Observe that \( \xrightarrow{\text{P}} \) is deadlock-free, i.e., for each configuration \( \gamma \in \Gamma \), there is at least one configuration \( \gamma' \in \Gamma \) such that \( \gamma \xrightarrow{\text{P}} \gamma' \).

### 4.3 Adding Probabilities: PTSO

We define the Markov Chain \( [\mathcal{P}]^{\text{MC}} = (\Gamma_p, M_p) \). The set \( \Gamma_p \) of configurations is defined as above. The probability matrix \( M_p \) is defined as the composition of two probability distributions: (i) the \textit{process} probability distribution \( M_{\text{proc}} \) (ii) the \textit{update} probability distribution \( M_{\text{update}} \) which add probabilities to the process transition relation \( \xrightarrow{\text{proc}} \), and the update transition relation \( \xrightarrow{\text{update}} \) respectively.

**The Process Probability Distribution: the Scheduler** At each program step (\( \xrightarrow{\text{P}} \)), a process is selected for execution according to a probability given by the scheduler. In a configuration \( \gamma \), the scheduler selects an enabled process \( p \in \text{enab}(\gamma) \) with a probability that reflects the relative weight of \( p \) compared to those of the other enabled processes, \( \text{Rweight}(\gamma)(p) \):

\[
\text{Rweight}(\gamma)(p) = \begin{cases} 
0 & \text{if } p \in \text{disab}(\gamma) \\
\frac{\text{Sched}(p)}{\sum_{p' \in \text{enab}(\gamma)} \text{Sched}(p')} & \text{if } p \in \text{enab}(\gamma)
\end{cases} \quad (1)
\]

This gives the probability that \( p \) to execute in the next step from \( \gamma \). For configurations \( \gamma \) and \( \gamma' \), with \( \gamma \xrightarrow{\text{proc}} \gamma' \), we define \( M_{\text{proc}}(\gamma, \gamma') := \text{Rweight}(\gamma)(p) \).

In other words, we move from \( \gamma \) to \( \gamma' \) with a probability that is given by the relative weight of \( p \) in \( \gamma \). We define \( M_p(\gamma, \gamma') := 0 \) if \( \gamma \not\xrightarrow{\text{proc}} \gamma' \). To account for

\(^8\) The latter transition is not strictly needed, but it is included for technical convenience.
the case where all the processes are disabled in $\gamma$, we define $M_{\text{proc}}(\gamma, \gamma) := 1$ if $\gamma$ is disabled.

**Faithfulness** Our model uses a scheduling policy that assigns a fixed scheduling weight, $\text{Sched}(p)$, to each process $p$ in the system. This is a case of memoryless scheduling, i.e., the probability distribution over processes does not depend on the execution history. However, we can relax this constraint to allow for any scheduling policy that satisfies the faithfulness condition:

$$\forall p \in \text{Procs} \ Rweight(\gamma)(p) = 0 \iff p \in \text{disab}(\gamma)$$

In words, at each step, each enabled process should be scheduled with non-zero probability. A scheduler that assigns scheduling weights such that the above condition holds is said to be a faithful scheduler.

**Schedulers with memory** The above criterion allows for schedulers that are more refined as compared to the memoryless scheduler. As an example, on implementations of TSO, processes are often scheduled for multiple consecutive steps since unnecessary context switching wastes processor resources. To reflect this detail, we can consider a scheduler that assigns a higher probability to the previously scheduled process, $p_{\text{prv}}$. For some choice of constant weights, $\text{Sched}$, we can define a new choice of weights $\text{Sched}'$ where $\lambda > 1$ is some parameter.

$$\text{Sched}'(p) = \text{Sched}(p) \text{ if } p \neq p_{\text{prv}} \text{ and } \lambda \cdot \text{Sched}(p) \text{ otherwise}$$

In this case, $p_{\text{prv}}$ is re-scheduled with a weight which is larger by a factor of $\lambda$. A larger $\lambda$ implies a stronger tendency to re-schedule a process. This scheduling policy still satisfies faithfulness. One can extend this by formulating more intricate policies, e.g. ones that account for $k$ previous steps.

To better illustrate the concerns and challenges of verification, we continue to adopt the simple (memoryless) scheduler proposed earlier. However, we emphasize that our results extend to faithful schedulers.

**The Update Probability Distribution: the Memory update policy** Between the process steps, pending messages from the store buffers are propagated to the shared memory (the update transition). The details of this write propagation are implementation-specific, with policies tuned towards system performance. Classical TSO models this update propagation non-deterministically. We, on the other hand, consider a probabilistic update policy. In a similar manner to the scheduling probabilities, the update probability distribution defines the probability by which a configuration $\gamma$ reaches another configuration $\gamma'$ through an update step ($\rightarrow_{\text{update}}$). Recall that an update step consists of a sequence of (single) update operations. The number of possible update sequences from $\gamma$ is finite since the sizes of each buffer is finite. In our model, we assume that the update distribution is the uniform distribution over all possible update sequences. We note that starting from $\gamma$, different update sequences can lead to the same configuration $\gamma'$. The reason is that different shufflings of the selected suffixes
(see Sec. 4.2) may lead to the same memory state. To reflect this, for configurations $\gamma$ and $\gamma'$, we define $M_{\text{update}}(\gamma, \gamma') := |\{\alpha \mid \gamma \xrightarrow{\alpha} \gamma'_{\text{update}}\}| / |\{\alpha \mid \exists \gamma'' \cdot \gamma \xrightarrow{\alpha} \gamma''_{\text{update}}\}|$, i.e. the fraction of update sequences that lead to the configuration $\gamma'$.

**Left-Biasedness** Though we adopt a specific update distribution, we provide a generic condition on that update policy that is sufficient for our results to hold. We call this the left-biasedness property. Here we provide an intuitive description of left-biasedness and defer the formal definition to Sec. 8.

Intuitively, left-biasedness requires that for sufficiently large configurations, the probability that the configuration size reduces in a single $\rightarrow_P$ step is strictly greater than $p$ for some $p > \frac{1}{2}$. Left-biasedness allows a wide class of more refined scheduling policies, e.g., where no message propagation is performed when the number of messages is smaller than a certain value, or where only the messages inside the buffers of some (probabilistically selected) processes are propagated.

Though our results apply more generally to models characterized by faithfulness (scheduler policy), and left-biasedness (update policy), we continue to adopt the fixed-weight (memoryless) scheduler and uniform update policy for reasons described above.

**The Full Probability Distribution.** We combine the process and update probability distributions, to derive the probability matrix $M_P$, and thus obtain the Markov chain $[P]^{\text{MC}}$. Consider configurations $\gamma$ and $\gamma'$ where $\gamma \rightarrow_P \gamma'$. Let $\gamma''$ be the unique configuration such that $\gamma \rightarrow_{\text{proc}} \gamma'' \rightarrow_{\text{update}} \gamma'$. Then, we define $M_P(\gamma, \gamma') := M_{\text{proc}}(\gamma, \gamma'') \cdot M_{\text{update}}(\gamma'', \gamma')$.

Lemma 1 $M_P$ is a prob. distribution on $\Gamma_P$; hence, $[P]^{\text{MC}}$ is a Markov chain.

5 PTSO: Concepts and Properties

Now, we intuit some concepts underlying Probabilistic TSO and its properties.

**PTSO Refines Classical TSO.** After introducing $[P]^{\text{TS}}$ and $[P]^{\text{MC}}$ in Sec. 4, we s.t. they are closely related; $[P]^{\text{TS}}$ is the underlying transition system of $[P]^{\text{MC}}$.

Lemma 2 $([P]^{\text{MC}})^\downarrow = [P]^{\text{TS}}$ for any program $P$.

In particular, this means that the PTSO system $[P]^{\text{MC}}$ is a refinement of $[P]^{\text{TS}}$: a behaviour is observed in $[P]^{\text{TS}}$ iff it is seen in $[P]^{\text{MC}}$ with non-zero probability. Whenever the context is clear, we write $P$ instead of $[P]^{\text{TS}}$, $[P]^{\text{MC}}$.

**Label Reachability.** We formulate our verification problems in terms of reachability to instruction labels. To simplify the notation, we identify a label $l \in $Lbl$ with the set $\Gamma_P$ of configurations in which $l$ occurs. We say that “$l$ is reachable” rather than “$l \in \Gamma_P$ is reachable”, and write $l$ instead of $\{\gamma \in \Gamma_P \mid l \in \gamma\}$. In [13,12] the authors show that label reachability from a plain configuration is decidable. The following lemma, generalizes this to the case where the source
configuration need not be plain and destination can be a particular plain configuration.

**Lemma 3** For a program $P$, a configuration $\gamma \in \Gamma_P$, and a plain configuration $\gamma' \in \Gamma_P^{\text{plain}}$, it is decidable whether $\gamma \xrightarrow{\ast} P \gamma'$.

Extending this, we have Lemma 4: we can query whether $\gamma \xrightarrow{\ast} P \gamma'$ for each $\gamma' \in \Gamma_P^l \cup \Gamma_P^{\text{plain}}$. Decidability of Lemma 4 follows since $\Gamma_P^{\text{plain}}$ is finite and the subroutine is decidable by Lemma 3.

**Lemma 4** For a program $P$, a configuration $\gamma \in \Gamma_P$, and a label $l \in \text{Lbl}_P$, it is decidable whether $\gamma \xrightarrow{\ast} P l$.

### 5.1 Left-Orientedness and Attractors

We show that the set of plain configurations $\Gamma_P^{\text{plain}}$ set has an attractor property in the sense of [16]. In our setting, this means that any run of $[P]^{\text{MC}}$ almost surely visits $\Gamma_P^{\text{plain}}$ infinitely often.

**Small and large configurations** To arrive at this result, we consider a generalization of plain configurations, called small configurations, denoted $\Gamma_P^{\text{small}}$. $\Gamma_P^{\text{small}}$ consists of configurations with a small number of messages inside their buffers. Concretely, a configuration $\gamma$ is small if $|\gamma| \leq 4$, i.e., the total number of messages inside the buffers does not exceed 4. $^9$ We define the set of large configurations by $\Gamma_P^{\text{large}} := \Gamma_P - \Gamma_P^{\text{small}} = \Gamma_P^{\geq 5}$. We show that the Markov chain $[P]^{\text{MC}}$ is left-oriented in the sense of [42]. That is, for any large configuration $\gamma \in \Gamma_P^{\text{large}}$, the expected change in configuration size for a single $\to P$ step is negative.

**An illustrative example** We explain the update probability distribution through the code snippet on the right. To begin with let us only consider the process on the left ($\text{procL}$). It executes an infinite loop, writing 1 to variable $x$. Let us consider the evolution of the buffer-sizes of procL, i.e. the number of $(x, 1)$ messages in the procL-buffer. Assume that on reaching label 0, procL has 6 messages in its buffer. The $\to \text{proc}$ step consists of a process transition, $\to \text{proc}$ followed by an update transition, $\to \text{update}$. In the $\to \text{proc}$ step, the write increases the size of the buffer by one, thus obtaining a buffer of size 7. Following this the $\to \text{update}$ step may push any number of messages to the memory. Since the update policy chooses uniformly amongst possible update sequences, the resulting configuration has one amongst $\{0, \ldots, 7\}$ messages in the procL-buffer, each occurring with an equal probability of $1/8$. The next $\to \text{proc}$ step (a $\text{goto}$), does not change the buffer size, but the $\to \text{update}$ step can still propagate messages. The reasoning for the next steps follows similarly.

---

$^9$ This value is an artifact of the probabilistic policies we have adopted in Sec. 4
Comparison with other notions of fairness At each $\rightarrow_{\text{proc}}$ step at most one message is added to the process buffers (when the process performs a write), however in the following $\rightarrow_{\text{update}}$ can still remove large number of messages. Hence, from sufficient large configuration sizes, the system has a tendency to move towards configurations with smaller buffer sizes. Formally, we prove the following lemma, using the left-orientedness property mentioned earlier.

Lemma 5  $\text{Prob}_P (\gamma \models \Box \Diamond \Gamma^\text{plain}_P) = 1$ for all configurations $\gamma \in \Gamma_P$.

For the above example, PTSO guarantees that the process on the right ($\text{procR}$) eventually reads value 1 into register $a$. This follows since in a plain configuration, the buffer of $\text{procR}$ is empty and hence it can read the value from the memory - this happens almost surely. We highlight that other notions of fairness such as strong fairness in process scheduling (discussed in [27]) as well as memory fairness [26], cannot provide this guarantee. In particular, memory fairness from [26], would consider the execution which exactly alternates writes of both processes but $\text{procR}$ reads before its own write is pushed memory to be fair and hence permissible.

$$x = 1 \quad x = 2 \quad a = x \quad // \quad 2 \quad x = 1 \quad x = 2 \quad a = x \quad // \quad 2 \quad x = 1 \quad \cdots$$

B-Plain Configurations We can refine our analysis of the attraction property enjoyed by the set $\Gamma^\text{plain}_P$ of plain configurations. We consider a subset of $\Gamma^\text{plain}_P$ which we call the set of bottom plain configurations, (or B-plain configurations, for short), denoted $\Gamma^\text{Bplain}_P$. Intuitively, a B-plain configuration is a member of a bottom strongly connected component in the graph of plain configurations. Formally, a configuration $\gamma \in \Gamma_P$ is said to be B-plain if (i) $\gamma \in \Gamma^\text{plain}_P$, and (ii) for any $\gamma' \in \Gamma^\text{plain}_P$, if $\gamma \rightarrow_{\text{P}} \gamma'$ then $\gamma' \rightarrow_{\text{P}} \gamma$. Since any run of the system almost surely visits the set of $\Gamma^\text{plain}_P$ infinitely often, it will also almost surely visit a B-plain configuration infinitely often.

Lemma 6  $\text{Prob}_P (\gamma \models \Box \Diamond \Gamma^\text{Bplain}_P) = 1$ for all configurations $\gamma \in \Gamma_P$.

6 Qualitative (Repeated) Reachability

| Given: a program $P$, a configuration $\gamma_{\text{init}} \in \Gamma_P$, a label $l \in \text{Lbl}_P$ |
|QUAL_REACH: Determine whether $\text{Prob}_P (\gamma_{\text{init}} \models \Diamond l) = 1$ |
|QUAL_REP_REACH: Determine whether $\text{Prob}_P (\gamma_{\text{init}} \models \Box \Diamond l) = 1$ |

In this section, we perform qualitative reachability analysis for PTSO. Given a program $P$, configuration $\gamma_{\text{init}}$, and label $l$, we check whether a $\gamma_{\text{init}}$-run almost surely reaches $l$. We also consider qualitative repeated reachability, where, we ask whether a $\gamma_{\text{init}}$-run repeatedly visits $l$ (visits $l$ infinitely often) w.p. 1. We also consider almost-never variants of the problems, where we check whether the probabilities are 0 rather than 1. We prove that these problems are decidable, and have non-primitive-recursive complexities.
6.1 Almost-Sure Reachability

The qualitative reachability problem, QualReach, is defined above. The algorithm in Figure 3 solves QualReach by analyzing the transition system $[P]^TS$, the underlying transition system of PTSO. If $l$ occurs in $\gamma_{\text{init}}$ then the property trivially holds, and hence we answer positively. Otherwise, the algorithm considers a new program $P'$ obtained by replacing the statement labeled $l$ by a new statement that makes $P'$ terminate immediately if $l$ is reached. Let $p \in \text{Proc}$ be the unique process such that $l \in \text{Lbl}_p$. We define $P \ominus l := \langle \text{Proc} - \{p\} \cup \{p'\}, \text{Sched} \rangle$ where $p'$ is a fresh process derived from $p$ by replacing $\text{stmt}(l)$ by $\text{goto}_{\text{term}}$ for a fresh label $\text{goto}_{\text{term}} \notin \text{Lbl}_P$ and adding a $\text{term}$ at label $\text{term}_{\text{new}}$. The remaining instructions of $p'$ are identical to $p$.

The loop on line 3 cycles through the (finite) set of plain configurations. For each plain configuration $\gamma$ from the original program $P$, we check: (i) Whether $\gamma$ is reachable from the initial configuration $\gamma_{\text{init}}$ in $P'$. By the construction of $P'$, this is equivalent to checking whether $\gamma$ is reachable from $\gamma_{\text{init}}$ in $P$ without observing label $l$. (ii) Whether it can reach the label $l$. If the answer to (i) is yes, and the answer to (ii) is no, then we have found a finite path $\pi$ in $P$ that starting from $\gamma_{\text{init}}$, without visiting $l$, reaches configuration $\gamma$ from which $l$ is not reachable. This implies that $\text{Prob}_P(\gamma_{\text{init}} \models \Diamond l) < 1$. If none of the plain configurations satisfy the condition, then each plain configuration $\gamma$ reachable from $\gamma_{\text{init}}$ has a path to $l$. Now by the attractor lemma, any run will almost surely visit $\Gamma_{\text{plain}}$ infinitely often and by the fairness property of Markov chains, it almost surely visits $l$.

6.2 Almost-Sure Repeated Reachability

For almost-sure repeated reachability we are interested in determining whether the $\gamma_{\text{init}}$-runs visit $l$ infinitely often with probability 1. The algorithm for this is similar to the case for almost-sure reachability: we check whether $\exists$ a plain configuration $\gamma$ that satisfies $\gamma_{\text{init}} \models \pi \Rightarrow \gamma$ and $\neg (\gamma \models \pi \Rightarrow l)$, in which case we return false. The difference is that we do not need to transform the program as in the case of almost-sure reachability. Details are in the supplementary material.

6.3 Almost-Never (Repeated) Reachability

The almost-never variants of the (repeated) reachability problems, NeverQualReach resp. NeverQualRepReach, ask whether the probabilities equal to 0 rather than 1. The solution to NeverQualReach is straightforward, since $\text{Prob}_P(\gamma_{\text{init}} \models \Diamond l) = 0$ iff $\neg (\gamma_{\text{init}} \models \pi \Rightarrow l)$. On the other
Given: a program $P$, a configuration $\gamma_{\text{init}} \in \Gamma_P$, a label $l \in \text{Lbl}_P$

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NEVER_QUAL_REACH</strong></td>
<td>Determine whether $\text{Prob}<em>P(\gamma</em>{\text{init}} \models \emptyset l) = 0$</td>
</tr>
<tr>
<td><strong>NEVER_QUAL_REP_REACH</strong></td>
<td>Determine whether $\text{Prob}<em>P(\gamma</em>{\text{init}} \models \Box \emptyset l) = 0$</td>
</tr>
</tbody>
</table>

hand, the **NEVER_QUAL_REP_REACH** problem requires a search over B-plain configurations $\gamma$ satisfying $\gamma_{\text{init}} \xrightarrow{*_P} \gamma \xrightarrow{*_P} l$. Due to space constraints, we defer the algorithm and proofs to the appendix.

### 6.4 Decidability and Complexity

The algorithms can be effectively implemented since (i) $\Gamma_P^{\text{plain}}$ is finite; and (ii) the conditions of the for-loops and if-statements can be checked effectively, as implied by Lemma 4. This gives Theorem 1. Theorem 2 is proved through reductions from the reachability problem under the classical (non-probabilistic) TSO semantics [19]. The non-primitive-recursive lower bounds follow from the corresponding result for reachability of classical TSO.

**Theorem 1.** **QUAL_REACH,** **QUAL_REP_REACH,** **NEVER_QUAL_REACH,** **NEVER_QUAL_REP_REACH** are all decidable.

**Theorem 2.** **QUAL_REACH,** **QUAL_REP_REACH,** **NEVER_QUAL_REACH,** **NEVER_QUAL_REP_REACH** all have non-primitive-recursive complexities.

### 7 Quantitative (Repeated) Reachability

In this section we discuss *quantitative* reachability problems for PTSO. In contrast to qualitative analysis from Sec. 6, the task here is to *compute* the actual probability. We are not able to compute the probabilities exactly, but we can approximate the probability with an arbitrary degree of precision.

#### 7.1 Approximate Quantitative Reachability

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>QUANT_REACH</strong></td>
<td>Determine $\theta$ s.t. $\text{Prob}<em>P(\gamma</em>{\text{init}} \models \emptyset l) \in [\theta, \theta + \varepsilon]$</td>
</tr>
<tr>
<td><strong>QUANT_REP_REACH</strong></td>
<td>Determine $\theta$ s.t. $\text{Prob}<em>P(\gamma</em>{\text{init}} \models \Box \emptyset l) \in [\theta, \theta + \varepsilon]$</td>
</tr>
</tbody>
</table>

In the approximate quantitative reachability problem, **QUANT_REACH**, given a precision parameter $\varepsilon$, we are interested in determining an approximation $\theta$ satisfying $\theta \leq \text{Prob}_P(\gamma_{\text{init}} \models \emptyset l) \leq \theta + \varepsilon$.

The algorithm in Fig. 4 solves the problem by successively improving the approximation at each iteration until we are within $\varepsilon$-precision of the exact value.
The algorithm maintains two variables: \text{PosApprx} (positive approximation) is an under-approximation of the probability with which \(l\) is reachable from \(\gamma_{\text{init}}\), and \text{NegApprx} (negative approximation) is an under-approximation of the probability with which \(l\) is not reachable from \(\gamma_{\text{init}}\). \text{PosApprx} serves as a lower bound on \(\theta\), while, \(1 - \text{NegApprx}\) serves as an upper bound: \(\text{PosApprx} \leq \theta \leq 1 - \text{NegApprx}\).

**Algorithm: QuantReach**

**Input:** \(P\): program; \(\gamma_{\text{init}} \in \Gamma_{P}\): configuration; \(l \in \text{Lbl}_{P}\): label; \(\varepsilon \in \mathbb{R}^{>0}\): precision.

**Var**

\(\text{PosApprx}, \text{NegApprx} \in \mathbb{R}\): approximations, \(\text{waiting} \in (\Gamma_{P} \times \mathbb{R})^{*}\): queue

1. \(\text{PosApprx} := 0; \text{NegApprx} := 0; \text{waiting} := (\gamma_{\text{init}}, 1)\)

2. while \(\text{PosApprx} + \text{NegApprx} < 1 - \varepsilon\) do

3. \(\langle \gamma, \phi \rangle := \text{head}(\text{waiting}); \text{waiting} := \text{tail}(\text{waiting})\)

4. if \(l \in \gamma\) then \(\text{PosApprx} := \text{PosApprx} + \phi\)

5. else if \((\gamma \rightarrow_{P} l)\) then \(\text{NegApprx} := \text{NegApprx} + \phi\)

6. else

7. for each \(\gamma'\) with \(\gamma \rightarrow_{P} \gamma'\) do \(\text{waiting} := \text{waiting} \cdot (\gamma', \phi \cdot M_{P}(\gamma, \gamma'))\)

8. return \(\text{PosApprx}\)

**Fig. 4.** The quantitative reachability algorithm.

The algorithm iteratively improves these approximations until we reach a point where their sum is within \(\varepsilon\) from 1 (line 4). In such a case, the desired value of \(\theta = \text{PosApprx}\) is an \(\varepsilon\)-precise approximation.

To calculate the approximations, the algorithm performs forward reachability analysis starting from the initial configuration \(\gamma_{\text{init}}\). It generates the set of \(\gamma_{\text{init}}\)-paths in a breadth-first manner, using the \text{waiting} FIFO queue. For each generated path \(\pi\) it also calculates the probability of \(\pi\). Instead of the whole path \(\pi\), \text{waiting} only stores the last configuration, \(\gamma\), of \(\pi\) and the probability of \(\pi\), \(\phi\), as a pair \(\langle \gamma, \phi \rangle\).

The approximation variables are initialized (line 3) to zero, and \text{waiting} queue is initialized to contain a single pair, \(\langle \gamma_{\text{init}}, 1 \rangle\), representing the initial configuration \(\gamma_{\text{init}}\) (which occurs with probability one). The while-loop executes until we achieve the desired precision. At each iteration, we check whether we already have reached the desired precision. If not, the algorithm pops the pair \(\langle \gamma, \phi \rangle\) from the \text{waiting}-queue. There are three possibilities depending on \(\gamma\):

1. If \(l \in \gamma\) (if-branch, line 6), the current path reaches \(l\) and, consequently, we increment \text{PosApprx} by \(\phi\), the weight of the current path.
2. If \(l\) is not reachable from \(\gamma\) (else-if branch, line 7), the measure of runs that reach \(l\) starting from \(\gamma\) is zero, and hence we increment \text{NegApprx} by \(\phi\).
3. If neither of the above hold (line 10), the current path needs to be explored further, we enqueue all successors \(\gamma'\) of \(\gamma\) into the queue. The probability of the new path to \(\gamma'\) is \(\phi \cdot M_{P}(\gamma, \gamma')\).

To show correctness of the algorithm, let \(\text{PosApprx}^{(i)}\) and \(\text{NegApprx}^{(i)}\) represent the value of \text{PosApprx} and \text{NegApprx} prior to performing the \(i^{th}\) iteration. We show that in the limit as \(i \rightarrow \infty\), the value of \(\text{PosApprx}^{(i)} + \text{NegApprx}^{(i)}\) tends to 1. Technically this follows by Lemma 5. By this lemma, any \(\gamma_{\text{init}}\)-run almost surely either (i) reaches a plain configuration from which \(l\) is not reachable, or
(ii) repeatedly reaches a plain configuration from which l is reachable. In case (ii) it will almost surely reach l. This implies that $\text{Prob}_P(\gamma_{\text{init}} \models (\Diamond(l \lor \neg\exists l))) = 1$, i.e., an $\gamma_{\text{init}}$-run will almost surely either reach l or reach a configuration from which l is not reachable, implying that $\text{PosApprx}^{(i)} + \text{NegApprx}^{(i)}$ tends to 1. Finally, by Lemma 4 we can effectively check the condition of the if-statement, and hence the algorithm terminates.

The correctness of the approximation on termination follows by the property that $\text{PosApprx}^{(i)}$ and $\text{NegApprx}^{(i)}$ are under-approximations of the reach and non-reach probabilities. This follows from the following invariants:

$\text{PosApprx}^{(i)} \leq \text{Prob}_P(\gamma_{\text{init}} \models \Diamond l)$ \hspace{1cm} $\text{NegApprx}^{(i)} \leq \text{Prob}_P(\gamma_{\text{init}} \models \Diamond\forall\neg l)$

$\text{Prob}_P(\gamma_{\text{init}} \models \Diamond l) \leq 1 - \text{Prob}_P(\gamma_{\text{init}} \models \Diamond\forall\neg l)$

$\text{PosApprx}^{(i)} + \text{NegApprx}^{(i)} > 1 - \varepsilon$ holds on termination

These imply that, on termination, $\text{PosApprx}$ is within $\varepsilon$-precision of $\theta$.

**Theorem 3.** $\text{Quant \_ Reach}$ is solvable.

### 7.2 Approximate Quantitative Repeated Reachability

In the case of the approximate quantitative repeated reachability problem, we are interested in approximating the probability of visiting a given label $l$ infinitely often. We develop an algorithm that uses an iterative approximation scheme similar to the reachability case. We defer full details of this algorithm to the supplementary material and instead give an intuitive explanation on how it differs from Sec. 7.1.

This algorithm too maintains approximations $\text{PosApprx}$ and $\text{NegApprx}$ and iteratively narrows the error margin until it is smaller than $\varepsilon$. The main difference is in the condition at line 6 of Figure 4. In the case of reachability the lower estimate $\text{PosApprx}$, is increased when $l \in \gamma$. In the repeated reachability case, this is not sufficient; we need to ensure that there is no state $\gamma'$ that is reachable from the current state $\gamma$ and such that $l$ is not reachable from $\gamma'$. The existence of such a $\gamma'$ implies existence of a non-zero measure continuation of the current run in which $l$ is not reached infinitely often. Hence, the conditional of the if-statement is modified to: $\forall \gamma' \in \text{BPlain} \Rightarrow (\gamma \xrightarrow{p} \gamma') \Rightarrow (\gamma' \xrightarrow{p} l)$.

We note that naively we would have to check the above condition for all configurations $\gamma' \in \Gamma_P$, which is infeasible since $\Gamma_P$ is an infinite set. We address this by using Lem. 6, which shows that runs from all configurations eventually reach a B-plain configuration. Hence it is sufficient to only check the condition for the (finitely many) B-plain configurations, which are precomputed in $\text{BPlain}$.

**Theorem 4.** $\text{Quant \_ Rep \_ Reach}$ is solvable.

### 8 Expected Average Costs

In this section, we develop a cost model for concurrent programs where we assign a cost to the execution of each instruction, the goal begin to approximate the expected cost of runs that reach a given label.
8.1 Computing costs over runs

A cost function $\text{Cost} : \text{Lbl}_P \to \mathbb{N}^{>0}$ for program $P$ defines for each label $l \in \text{Lbl}_P$ the cost of executing the instruction at $l$. A particular way to define the function is to assign a cost to each instruction in the programming language, so that $\text{Cost}(l)$ depends only on $\text{stmt}(l)$ and not on $l$ itself. But we consider the general case. We extend $\text{Cost}$ to runs as follows. Consider configurations $\gamma = (\lambda, R, B, M)$ and $\gamma'$ such that $\gamma \rightarrow_P \gamma'$. If $\gamma \xrightarrow{\rho} P \gamma'$, for process $p$, then we define $\text{Cost}(\gamma, \gamma') := \text{Cost}(\lambda(p))$. In other words, it is the cost of the instruction executed by $p$. Recall from Sec. 4 that $p$ is unique and therefore the function is well-defined. If $\text{disab}(\gamma)$ or if $\neg (\gamma \rightarrow_P \gamma')$ then we define $\text{Cost}(\gamma, \gamma') := 0$. Consider a run $\rho \in \{\text{Runs}(\gamma) \mid \rho \models_P \Diamond^i l\}$, i.e. a $\gamma$-run that reaches $l$ for the first time at step $i$. We define $\text{Cost}(\rho)(l) = \sum_{1 \leq j \leq |i|-1} \text{Cost}(\rho[j], \rho[j+1])$, i.e., the sum of costs of all executed instructions along $\rho$ up to the first visit to $l$.

For a configuration $\gamma$, a label $l$, and a cost function $\text{Cost}$, we define a random variable $X_{\gamma, l, \text{Cost}} : \Omega \to \mathbb{R}$ over support $\Omega = \gamma \cdot R^\omega$ as follows:

$$X_{\gamma, l, \text{Cost}}(\rho) = \begin{cases} 0 & \rho \notin \{\text{Runs}(\gamma) \mid \rho \models_P \Diamond^i l\} \\ X_{\gamma, l, \text{Cost}}(\rho)(l) & \text{otherwise} \end{cases}$$

**Given:** program $P$, configuration $\gamma\_\text{init} \in \Gamma_P^{\text{plain}}$, cost function $\text{Cost} : \text{Lbl}_P \to \mathbb{N}^{>0}$, label $l \in \text{Lbl}_P$ s.t. $\gamma\_\text{init} \models \Diamond l$, precision value $\epsilon \in \mathbb{R}^+$

**Exp\_AVE\_Cost:** Determine $\theta$ s.t. $E(X_{\gamma\_\text{init}, l, \text{Cost}} \mid \gamma\_\text{init} \models \exists l) \in [\theta, \theta + \epsilon]$

The expected average cost problem $E(X_{\gamma, l, \text{Cost}})$ is defined as the expected cost of reaching $l$ from $\gamma$ and $E(X_{\gamma, l, \text{Cost}} \mid \gamma \models \exists l)$ as the conditional expectation over runs that reach $l$. If $\neg (\gamma \models_P \exists l) \text{ then the expected cost is not defined.}$ If however $\gamma \models_P \exists l$ then $E(X_{\gamma, l, \text{Cost}} \mid \gamma \models \exists l) = E(X_{\gamma, l, \text{Cost}}) / \text{Prob}_P (\gamma \models_P \\exists l)$, which follows since for the non-reaching runs, the cost is zero. We present the expected average cost problem, in the figure above, where we want to approximate $E(X_{\gamma, l, \text{Cost}} \mid \gamma \models \exists l)$ to $\epsilon$-precision.

8.2 Eagerness

Our solution to Exp\_AVE\_Cost relies on the fact that $\llbracket P \rrbracket^{MC}$ satisfies an eagerness property in the sense of [17]. In our setting, eagerness means that the probability of avoiding the target label $l$ decreases exponentially with the number of steps. Concretely, we show that there are two constants: the eagerness degree $E_P \in \mathbb{R}^{>0}$, and the eagerness threshold $\eta_P \in \mathbb{R}^{>0}$ satisfying the following:

$$\forall \gamma \in \Gamma_P^{\text{small}} \forall l \in \text{Lbl}_P \forall n \geq \eta_P \; \gamma \models_P \exists l \Rightarrow \text{Prob}_P (\gamma \models_P \Diamond^n l) \leq (E_P)^n$$

i.e. for $n \geq \eta_P$, the probability of avoiding $l$ during the first $n$ steps decreases exponentially with $n$. The following lemma forms the crux of this section.

**Lemma 7 (Eagerness Lemma)** $E_P$ and $\eta_P$ exist and are computable.
We devote this sub-section to give an overview of the proof of Lemma 7 (the formal proof is provided in the supplementary material). We consider the behavior of runs with respect to the small and large configurations, exploiting the fact that the runs of the system tend to gravitate towards the small configurations. However here we use a property, called left-biasedness (defined in Sec. 8.2), that is stronger than the left-orientedness property of Sec. 5.1.

To prove Lemma 7, we show that, for a small configuration \( \gamma \in \Gamma_P^{\text{small}} \), the runs from \( \gamma \) satisfy the following three properties with a high probability: (i) they make their first return to \( \Gamma_P^{\text{small}} \) within a small number of steps, (ii) they return to \( \Gamma_P^{\text{small}} \) multiple times, within a small number of steps, and (iii) if they eventually reach 1 then they will do that within a few steps. We collect these results to obtain the proof of Lemma 7.

Gravity: First Return We recall that buffer sizes can increase by at most one during process transitions, and that any number of messages can be flushed to the memory during an update transition (Sec. 4 and Sec. 5.1). Based on this, we show left-biasedness, defined as follows:

**Left-biasedness** \( \forall \gamma \in \Gamma_P^{\text{large}} \) the probability of moving from \( \gamma \) to a smaller configuration is bounded below by \( 2/3 \) and that of moving to a larger configuration is bounded above by \( 1/3 \), regardless of \( P \).

Using left-biasedness, we show that the set \( \Gamma_P^{\text{small}} \) has a gravity property, namely, a run starting from a small configuration will, with a high probability, return to the set \( \Gamma_P^{\text{small}} \) (for the first time) within a few number of steps. Formally, we define the gravity parameter \( \mathcal{G}_P \) as follows: \( \hat{q} := 2/3, \hat{p} := 1/3, \) and \( \mathcal{G}_P := 2\sqrt{\hat{q} \cdot \hat{p}} = \frac{2\sqrt{2}}{3} \). We prove the following lemma.

**Lemma 8 (Gravity Lemma)** \( \text{Prob}_P (\gamma \models P \bigcirc \ominus^n \Gamma_P^{\text{small}}) \leq (\mathcal{G}_P)^n \), for all \( \gamma \in \Gamma_P^{\text{small}} \) and all \( n \in \mathbb{N} \).

The lemma states that, starting from a small configuration, the probability that a run avoids \( \Gamma_P^{\text{small}} \) in the next \( n \) steps decreases exponentially with \( n \).

Multiple Revisits Notice that the gravity lemma is concerned with the first return to the set of small configurations. We will now apply this argument repeatedly to conclude that, with high probability, multiple re-visits to small configurations take place “quickly”. That is, the set of runs starting from \( \Gamma_P^{\text{small}} \) and frequently re-visiting \( \Gamma_P^{\text{small}} \) has a high measure. To formalize these arguments, we make the following definition. For \( m, n : 1 \leq m \leq n \), we define \( \text{Visit}_P (n, m) \) to be the set of runs that visit the set \( \Gamma_P^{\text{small}} \) exactly \( m \) times in their first \( n - 1 \) steps\(^{10}\). We use the \( \text{Visit} \) predicate to partition the set of \( \gamma \)-runs, depending on how often they return to \( \Gamma_P^{\text{small}} \) during their first \( n \) steps. We distinguish these as

---

\(^{10}\) For technical convenience, we use \( n - 1 \) instead of \( n \) in the definition of \( \text{Visit} \). This allows us to avoid some corner cases in the proofs.
Sporadic-Runs (S-Runs): runs that visit the $\Gamma^\text{small}_P$ sporadically during their first $n$ steps, and Frequent-Runs (F-Runs): runs that visit $\Gamma^\text{small}_P$ frequently during their first $n$ steps. We will derive a constant $\nu \in \mathbb{N}$ (see below) that delineates the border between these sets. We formally define:

$$\text{SRuns}(\gamma)(n) := \bigcup_{1 \leq m \leq \left\lfloor \frac{n}{\nu} \right\rfloor} \{ \rho \in \text{Runs}(\gamma) \mid \rho \models \text{Visit}_P(n,m) \}$$

$$\text{FRuns}(\gamma)(n) := \bigcup_{1 \leq m \leq \left\lfloor \frac{n}{\nu} \right\rfloor} \{ \rho \in \text{Runs}(\gamma) \mid \rho \models \text{Visit}_P(n,m) \}$$

![Fig. 5.](image.png)

**Fig. 5.** Figure depicting configuration sequences of S, F and D runs. Green dots represent small configurations, blue dots represent large configurations. All runs start in a small (plain) configuration. Within the first $n$ configurations: the S-run visits $\Gamma^\text{small}_P$ at most $\left\lfloor \frac{n}{\nu} \right\rfloor$ times, the F, D runs visit $\Gamma^\text{small}_P$ at least $\left\lfloor \frac{n}{\nu} \right\rfloor + 1$ times. A D-run is a special case of an F-run which does not visit label $l$ (red dot) in the first $n$ steps.

The value of $n/\nu$ distinguishes the S-Runs from the F-Runs. Our goal is to give an upper bound on the measure of the S-Runs. For a prefix path $\pi$ of length $n$, there are $(n-m)$ ways to choose the $m-1$ indices along $\pi$ at which $\Gamma^\text{small}_P$ is reached (since the run starts from $\Gamma^\text{small}_P$). Each of the $m-1$ path fragments between these indices represents one consecutive revisit of $\Gamma^\text{small}_P$. By Lemma 8, the measure of the set of such runs is bounded by $(G_P)^{n-m}$, giving

$$\text{Prob}_P (\text{SRuns}(\gamma)(n)) \leq \sum_{m=1}^{\left\lfloor \frac{n}{\nu} \right\rfloor} (n-1) \cdot (2+\sqrt{3} \cdot \nu)^{\left\lfloor \frac{1}{2} \right\rfloor}$$

under the condition that $4 \leq 2 \cdot \nu \leq n$. The second inequality is obtained through algebraic manipulations using $G_P = 2\cdot\sqrt{\frac{2}{3}}$. Define $f(x) := \sqrt{\frac{8}{3}} \cdot \left( \frac{x}{x-1} \right) \cdot (2+\sqrt{3} \cdot x)^{\left\lfloor \frac{1}{2} \right\rfloor}$. We have $f(150) = 0.986 < 1$. Hence, for parameter $\nu := 150$, defining $E_P^S := f(\nu)$, we have the following lemma, where the bound decays exponentially with $n$ since $E_P^S < 1$.

**Lemma 9 (S-Run Bound)** $\text{Prob}_P (\gamma \models \text{SRuns}(\gamma)(n)) \leq (E_P^S)^n$, for all $\gamma \in \Gamma^\text{small}_P$ and all $n$ such that $300 = 2 \cdot \nu \leq n$.

**Reaching the label l** We now turn our attention to the set of F-Runs. Our goal is to show that if an F-Run reaches $l$ then, with a high probability, it will reach $l$ "quickly". To that end, we consider the opposite scenario and introduce a subset of the F-Runs which we call Delayed Runs (D-Runs):

$$\text{DRuns}(\gamma)(l)(n) := \bigcup_{m=\left\lfloor \frac{n}{\nu} \right\rfloor + 1} \{ \rho \in \text{Runs}(\gamma) \mid \rho \models \Diamond^n (l \wedge \text{Visit}_P(n,m)) \}$$
A D-Run is an F-Run that *delays* its first visit to the label $l$ until the $n^{th}$ step for some $n$. We show that the measure of D-Runs decreases $n$ increases. Note that $l$ is reachable from all configurations from a path that ends at $l$. Therefore, we consider the set $\mathcal{A} := \{ \gamma \in \mathcal{I}_{SD}^{small} \mid \gamma \models_{P} \exists l \}$, of small configurations from which $l$ is reachable. We analyze how often a run starting from a small configuration, visits $\mathcal{A}$ before finally visiting the label $l$. For sets of configurations $G_1, G_2 \subseteq \mathcal{I}_{P}$, a run $\rho$, and $m \in \mathbb{N}$, we write $\rho \models G_1 \text{Before}^m G_2$ to denote that $\rho$ visits the set $G_1$ at least $m$ times before visiting $G_2$ for the first time. Notice

\[
\text{DRuns}(\gamma)(l)(n) \subseteq \bigcup_{m=\lfloor \frac{n}{2} \rfloor + 1}^{n} \{ \rho \in \text{Runs}(\gamma) \mid \rho \models_{P} \mathcal{A} \text{Before}^m l \} \tag{2}
\]

To upper bound the measure of D-Runs, we start by upper bounding the measure of the set $\{ \rho \in \text{Runs}(\gamma) \mid \rho \models_{P} \mathcal{A} \text{Before}^m l \}$, i.e. $\gamma$-runs making $m$ visits to $\mathcal{A}$ before visiting $l$. We consider the probability that a run from a small configuration $\gamma$ does visit $l$ before returning to $\gamma$. We can compute a $\mu$ such that

\[
0 < \mu \leq \min_{\gamma \in \mathcal{A}} \text{Prob}_{\gamma}(\gamma \models \bigcirc(l \text{Before}^1 \gamma)) \tag{3}
\]

Hence $\mu$ is a lower bound on the measure of runs that start from some configuration in $\gamma \in \mathcal{A}$ and visit $l$ before returning to $\gamma$. To obtain an upper bound on the measure of D-Runs, we show the following inequality:

\[
\text{Prob}_{\gamma}(\text{DRuns}(\gamma)(l)(n)) \leq \sum_{m=\lfloor \frac{n}{2} \rfloor + 1}^{n} \sum_{\gamma \in \mathcal{A}} (1 - \mu)^{\left\lfloor \frac{|A|}{\pi(l)} \right\rfloor} \leq \frac{|A|}{(1-\mu)(1-(1-\mu)^{|A|})} \cdot \left((1-\mu)^{\frac{n}{|A|}}\right)
\]

The first inequality follows from formulas 2 and 3, while the second is obtained through algebraic techniques. Define $\epsilon_{SD}^{\text{P}}$ such that $(1-\mu)^{\frac{1}{|A|}} < \epsilon_{SD}^{\text{P}} < 1$. Such an $\epsilon_{SD}^{\text{P}}$ is computable since $\nu, A, \mu$ are computable. Since $(1-\mu)^{\frac{1}{|A|}} < \epsilon_{SD}^{\text{P}}$, it follows that there is a natural number, denoted by $\eta_{SD}^{\text{P}}$, such that

\[
\left((1-\mu)^{\frac{1}{|A|}}\right)^n \leq (\epsilon_{SD}^{\text{P}})^n \text{ for all } n \geq \eta_{SD}^{\text{P}}.
\]

This gives the following lemma.

**Lemma 10 (D-Run Bound)** $\text{Prob}_{\gamma}(\text{DRuns}(\gamma)(l)(n)) \leq (\epsilon_{SD}^{\text{P}})^n$, for all $\gamma \in \mathcal{I}_{SD}^{\text{small}}$ and all $n \geq \eta_{SD}^{\text{P}}$.

**Proof of Lemma 7** We now give a sketch of the proof of the eagerness property.

Choose a value $\epsilon_{SD}^{\text{P}}$ such that, $\max(\epsilon_{SD}^{\text{P}}, \epsilon_{SD}^{\text{D}}) < \epsilon_{SD}^{\text{SD}} < 1$. From Lemma 9 and Lemma 10 it follows that for some constant $\eta_{SD}^{\text{P}}, \max(\eta_{SD}^{\text{P}}, 300)$, $\text{Prob}_{\gamma} (\gamma \models \bigcirc(n \text{)} l) \leq \left(\epsilon_{SD}^{\text{P}}\right)^n$, for all $n > \eta_{SD}^{\text{SD}}$ (sufficiently large). The final step is to extend the argument to the set of $\gamma$-runs that reach $l$ in $n$ or more steps (as required by Lemma 7).

\[
\text{Prob}_{\gamma} (\gamma \models \bigcirc(n \text{)} l) = \sum_{k=n}^{\infty} \text{Prob}_{\gamma} (\gamma \models \bigcirc(n \text{)} l) \leq \sum_{k=n}^{\infty} \left(\epsilon_{SD}^{\text{P}}\right)^k = \frac{(\epsilon_{SD}^{\text{P}})^n}{1-\epsilon_{SD}^{\text{P}}}
\]

Choose $\epsilon_{\text{P}}$, (exists since $\epsilon_{SD}^{\text{SD}} < 1$) such that $\epsilon_{SD}^{\text{SD}} < \epsilon_{\text{P}} < 1$. There exists an $\eta_{\text{P}}$ such that $\left(\epsilon_{\text{P}}\right)^n \leq (\epsilon_{\text{P}})^n$ for all $n \geq \eta_{\text{P}}$, and hence $\text{Prob}_{\gamma} (\gamma \models \bigcirc(n \text{)} l) \geq (\epsilon_{\text{P}})^n$ for all $n \geq \eta_{\text{P}}$ (sufficiently large). This gives us the result.
8.3 The Algorithm

Now we proceed to describe the algorithm. The goal is to approximate $E(X_{\gamma_{\text{init}}, \text{Cost}} \mid \gamma_{\text{init}} \models \exists \emptyset l)$. The scheme followed by the algorithm is similar to the quantitative section: it iteratively improves an approximations until it is $\varepsilon$-precise. However, the implementation is much more challenging since we need to maintain error margins on both the cost and the probabilities. It performs forward reachability analysis, starting from $\gamma_{\text{init}}$, and generating, successively longer $\gamma_{\text{init}}$-paths, in a breadth-first manner.

The variable \texttt{waiting} contains triples of form $\langle \gamma, \psi, \phi \rangle$ corresponding to $\gamma_{\text{init}}$-paths waiting to be analysed. For such a path $\pi$, $\gamma$ is the last configuration of $\pi$, $\psi$ is the cost of $\pi$, and $\phi$ is the probability of taking $\pi$. We initialize \texttt{waiting} to contain a triple corresponding to the empty path from $\psi$. paths waiting to be analysed. For such a path $\gamma_{\text{init}}$, we analyse and the $\pi_{\text{waiting}}$ are studied and the \texttt{iteration loop (line 10)}, \texttt{waiting} contains triples corresponding to paths of length $i$. At each loop iteration the triples in \texttt{waiting} are analysed and the triples for paths one step deeper are generated for the next iteration.

```
Algorithm: Solving Exp_Ave_Cost

Input: $\mathcal{P}$: program; $\gamma_{\text{init}} \in \mathcal{T}\mathcal{P}$: configuration $l \in \text{Lbl}_{\mathcal{P}}$: label with $\gamma_{\text{init}} \models \exists \emptyset l$; $\mathcal{L}$: label with $\gamma_{\text{init}} \models \exists \emptyset l$; $\mathcal{P}$: cost function; $\varepsilon \in \mathbb{R}^{>0}$: precision;
1 Var
2 waiting, waiting' $\in (\mathcal{I} \times \mathcal{R} \times \mathcal{R})^*$: queues;
3 CostApprx $\subseteq \mathbb{R}$: under-approximation of $E(X_{\gamma_{\text{init}}, \text{Cost}})$;
4 ProbApprx $\subseteq \mathbb{R}$: under-approximation of $\mathbb{P} \big( \gamma_{\text{init}} \models \exists \emptyset l \big)$;
5 CostError $\subseteq \mathbb{R}$: under-approximation of errors;
6 $k, n \in N$;
7 $k := \text{MaxCost}(\text{Cost})$; $n := 0$;
8 CostApprx := 0; ProbApprx := 0; waiting := $\langle \gamma_{\text{init}}, 0, 1 \rangle$;
9 CostError := $\frac{1}{(1 - \varepsilon)^{\gamma_{\text{init}}}}$; ProbError := $\frac{1}{1 - \varepsilon^{\gamma_{\text{init}}}}$;
10 repeat
11 $n := n + 1$; waiting' := $\emptyset$;
12 for $i = 1$ to $|\text{waiting}|$ do
13 $\langle \gamma, \psi, \phi \rangle := \text{waiting}[i]$;
14 if $l \in \gamma$ then
15 CostApprx := CostApprx + $\psi \cdot \phi$; ProbApprx := ProbApprx + $\phi$;
16 else
17 for all $\gamma': \gamma \rightarrow_{\mathcal{P}} \gamma'$ do
18 $\text{waiting}' := \text{waiting}' \cdot \langle \gamma', \psi + \text{Cost}(\gamma, \gamma'), \phi \cdot \mathcal{M}_{\mathcal{P}}(\gamma, \gamma') \rangle$;
19 CostError := CostError $\cdot \mathcal{E}_{\mathcal{P}}$; ProbError := ProbError $\cdot \mathcal{E}_{\mathcal{P}}$;
20 waiting := waiting';
21 until $(\text{CostApprx + CostError} \frac{\text{CostApprx}}{\text{ProbApprx + ProbError}} < \varepsilon) \wedge (\text{ProbError} > 0) \wedge (n \geq n_0)$;
22 return $\frac{\text{CostApprx + CostError}}{\text{ProbApprx + ProbError}}$;
```

Fig. 6. The expected average cost algorithm.

The iterations calculate increasingly precise approximations of $E(X_{\gamma_{\text{init}}, \text{Cost}})$, and of $\mathbb{P} \big( \gamma_{\text{init}} \models \exists \emptyset l \big)$, maintained in variables \texttt{CostApprx} and \texttt{ProbApprx}, respectively. We maintain two additional variables (\texttt{CostError} and \texttt{ProbError}) that help us to provide an upper bound on the estimation errors. Defining $\text{MaxCost}(\text{Cost}) := \max \{ \text{Cost}(l) \mid l \in \text{Lbl}_{\mathcal{P}} \}$, we explain the correctness of the algorithm with a number of invariants.
Lemma 11. The algorithm maintains the following invariants where invariants (1,2,5,6) hold for all \(i > 0\) and invariants (3,4) hold for all \(i \geq \eta P\).

1. \(\text{CostApprx}^{(i)} = \sum_{\rho \in \text{Runs}((\gamma_{\text{init}}) \mid \rho = \emptyset)^{\leq i}} \text{Cost}(\rho) \cdot \text{Prob}(\rho)\):
2. \(\text{ProbAppx}^{(i)} = \text{Prob}(\gamma_{\text{init}} \models \emptyset^\leq i)\):
3. \(\text{CostApprx}^{(i)} \leq E(X_{\gamma,1,\text{Cost}}) \leq \text{CostApprx}^{(i)} + \text{CostError}^{(i)}\):
4. \(\text{ProbAppx}^{(i)} \leq \text{Prob}(\gamma \models P \emptyset^0) \leq \text{ProbAppx}^{(i)} + \text{ProbError}^{(i)}\):
5. \(\text{CostError}^{(i)} = \max \{\text{Cost}(\rho) \cdot \text{Prob}(\rho)\} \geq \text{Cost}^{(i)} + \text{ProbError}^{(i)}\):
6. \(\text{ProbError}^{(i)} = \frac{\epsilon_p^i}{1 - \epsilon_p}\).

Invariants 5 and 6 imply that as \(i \to \infty\) \(\text{CostError}^{(i)}\) and \(\text{ProbError}^{(i)}\) tend to 0. Hence, \(\lim_{i \to \infty} \left(\frac{\text{CostApprx}^{(i)} + \text{CostError}^{(i)}}{\text{ProbAppx}^{(i)}} - \frac{\text{CostApprx}^{(i)} - \text{CostError}^{(i)}}{\text{ProbAppx}^{(i)} + \text{ProbError}^{(i)}}\right) = 0\) implying termination. Since \(n \geq \eta P\) when the algorithm terminates, by invariants 3 and 4 it follows that \(\text{CostApprx}^{(n)} \leq E(X_{\gamma,1,\text{Cost}}) \leq \text{CostApprx}^{(n)} + \text{CostError}^{(n)}\) and \(\text{ProbAppx}^{(n)} \leq \text{Prob}(\gamma \models P \emptyset^0) \leq \text{ProbAppx}^{(n)} + \text{ProbError}^{(n)}\). Combining these two inequalities and the termination condition of the algorithm, we get the following:

\[
\frac{\text{CostApprx}^{(n)}}{\text{ProbAppx}^{(n)} + \text{ProbError}^{(n)}} \leq \frac{E(X_{\gamma,1,\text{Cost}})}{\text{Prob}(\gamma \models P \emptyset^0)} \leq \frac{\text{CostApprx}^{(n)}}{\text{ProbAppx}^{(n)} + \text{ProbError}^{(n)}} + \epsilon.
\]

Hence on termination, \(\theta := \frac{\text{CostApprx}^{(n)}}{\text{ProbAppx}^{(n)} + \text{ProbError}^{(n)}}\) is within \(\varepsilon\)-precision of the true value, implying correctness of the algorithm. We get the following theorem.

Theorem 5. The above algorithm solves \(\text{Exp\_AVE\_COST}\).

9 Conclusions, Discussions, and Perspectives

We presented \(\text{PTSO}\), a probabilistic extension of the classical TSO semantics. We have shown decidability/computability results for a wide range of properties such as quantitative and qualitative reachability/repeated reachability and expected average costs. As far as we know, this is the first study of probabilistic verification for weak memory models, and opens many avenues for future work.

Refined Probability Distributions. For ease of presentation, we developed our results in the context of specific scheduling and update policies. However, we emphasize that our results carry-over to policies satisfying faithfulness and left-orientedness, which are fairly weak conditions. Hence we believe that developing more refined models that better capture behaviours of TSO implementations, using techniques such as parameter estimation, is interesting future work.

General Cost Models Similar can be said for cost models: our algorithm works for all cost functions such that the cost of a path is exponentially bounded by its length. In particular, developing cost models that closely mimic usage of processor resources, e.g., cost based on read from local store-buffer vs. read from memory, can be useful to gain a better understanding of the implementation.

Other Memory Models Finally, we are interested in extending our approach to other weak memory models such as RA/SRA, POWER, ARM.
References


Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the
source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.