Termination in Extended Probabilistic Threshold Automata

Mouhammad Sakr and Marcus Völp
mouhammad.sakr@uni.lu, marcus.voelp@uni.lu

SnT, Luxembourg University

Abstract. Scaling up distributed systems increases also the chance that some node ceases to operate correctly, which turns fault tolerance into an essential trait. To achieve fault tolerance, numerous distributed algorithms, including reliable broadcast and consensus, depend on threshold guards. A threshold guard can, for example, ensure that a process waits for a majority of its peers to acknowledge that they reached a certain state in the distributed algorithm, before the process makes any progress. Threshold automata are computational models that allow fully automated parameterized verification of single- and multi-round threshold-based distributed algorithms (FTDA), where often the number of processes and the proportion of faulty processes are parameters. However, due to the fact that such algorithms have to cope with faulty processes not answering or, more generally, behaving in an arbitrary potentially malicious manner, liveness must only depend on a subset of processes, while ideally all correct processes should be considered in the termination properties of such algorithm. In this paper, we present a novel reasoning technique for proving almost-sure termination in extended probablistic threshold automata, by detecting strongly connected components (SCCs) in extended probabilistic and in ordinary threshold automata to reduce almost-sure termination to reachability in a finite abstract system.

A personal note by Marcus

Dear Christel,
remember when we first met. Me coming from operating systems, it took us a while before we understood each other. Model checkers were always suspicious to me, in particular after you confirmed that variable declaration order can make a difference between getting results and waiting for millenia. Thanks for taking away this skepticism and for making me cherish the prospect of obtaining the highest assurance guarantees known today, just by pressing a button. Remember also, when we stumbled over a strongly connected component (SCC) in our characterization of conditional probabilities on the long run of our model of locks in coherent caches [1]. What started as a toy example, with a number of cores that was already outpaced by reality, turned into a scalable solution for core counts that we barely see today. In the present paper, Mouhammad and I return to SCCs to prove almost-sure termination in extended probablistic threshold automata. Thanks for leading the paths of so many researchers, including myself,

to not shy away from formal verification and model checking in particular.

I wish you all the best,
Marcus

1 Introduction

In large scale distributed systems, faults are inevitable and may lead to arbitrary and possibly intentionally malicious, that is, Byzantine behavior of nodes and the processes they execute. Consequently, the distributed algorithms that govern these systems must tolerate faults, whether they are accidental or caused by cyberattacks. Prominent examples of such fault-tolerant distributed algorithms (FTDAs) are reliable broadcast [9], Byzantine fault tolerant agreement [19], and the various variants of the two that currently form the dissemination and consensus layers of modern permissioned and permissionless blockchains [22,23]. Unfortunately, FTDAs are hard to get correct, so we need formal verification and tools to assist developers constructing them.

Threshold automata (TAs) are formal models of FTDAs that avoid the need to enumerate faulty behavior and that are able to characterize systems in a parametric manner. While the parameterized verification problem is generally undecidable [21], it is decidable for certain classes of systems [10, 11], including threshold automata [2, 12].

FTDAs, like the ones above, must on the one side be able to detect and recover from or, better, mask any behavior of faulty processes, including no response to requests at all, but also possibly long correct behavior patterns. Consequently, given a fault tolerance threshold t, where the actual number of faulty processes satisfies $f \leq t$, the algorithm must ensure progress with only $n - t$ processes, as it cannot depend on responses from potentially faulty ones. In addition, FTDAs must also ensure that no conflicting decisions are taken, be that binary consensus (e.g., whether to deliver a message) by following a simple majority (e.g., more than $n/2$), or agreement on a value by requiring a correct process in the intersection of any two quorums [18] that can take such a decision (e.g., $2Q - n \leq t + 1$ where Q denotes the quorum size). Threshold automata characterize such algorithms, by demanding that a threshold of processes are in a given state (e.g., of agreement), before progress can be made.

In this work, we deal with randomized systems, so we use probabilistic threshold automata (PTAs). One randomized binary consensus algorithm is Ben-Or's algorithm [4], see Algorithm 1 and its corresponding TA in Figure 1. It proceeds in rounds, where $n - t$ processes execute rounds in lock-step in an asynchronous manner. Each round is comprised of two phases. In the first, each process tries to identify a value $v \in \{0, 1\}$ that is supported by a majority. In the second, decision or ratification phase, processes finalize their decision if the value is proposed by at least $t + 1$ processes among the $n = 2t + 1$ processes. In case no such simple majority can be found, Ben-Or's algorithm causes some nodes to change their votes for the next round, repeating the agreement until eventually

probabilistically the system converges to a consensus value in one of the future rounds.

In this work, we lift two restrictions that were imposed on threshold automata during their verification, namely that cycles are not allowed and that coin tosses may only appear at the end of the automaton. With these restrictions in place, algorithms like Ben-Or's could only be verified under round-rigid adversarial schedules [6], which require all processes to complete the previous round r before any process can start round $r+1$. The algorithm as verified, would have to either wait for faulty processes to acknowledge this fact, or would have to prevent them from behaving correctly and push other correct processes beyond this boundary.

We do so by introducing a novel algorithm for detecting strongly connected components and apply it to prove termination in a system model for a network of an arbitrary number of deterministic threshold automata. The algorithm further detects strongly connected components and almost-sure termination (i.e., termination with probability one) in a system model for a network of an arbitrary number of probabilistic threshold automata.

We start by introducing our system model and the extension of probabilistic threshold automata (PTAs) that allows cycles in Section 2. In Section 3, we introduce a finite abstract domain for the shared variables of TAs and an abstraction based on parametric interval abstraction. Section 4 demonstrates how this new abstraction allows dropping the resilience condition, the function N that determines the number of processes to be modeled, and the exact number of processes in a configuration. Section 5 introduces our algorithm for detecting almost-sure termination and strongly connected components. Section 6 relates our work to the works of others, before we draw conclusions and highlight future work in Section 7.

2 System Model

In this section, we build upon the existing concept of probabilistic threshold automata (PTA) [6], extending the definition to allow shared variables to be reset, similar to [3]. We then show how to transform a PTA into a non-deterministic threshold automaton (TA), where non-determinism arises from multiple rules (transitions) being enabled simultaneously. We also define the semantics of an unbounded number of non-deterministic TAs running in parallel.

Definition 1. *A* probabilistic threshold automaton *(PTA) is a tuple* $PA = (L, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$ *where:*

- L *is a finite set of* locations.
- $\mathcal{I} \subseteq L$ *is the set of* initial locations.
- $\Gamma = \{x_0, \ldots, x_m\}$ *is a finite set of* shared variables *over* $\mathbb{N}_0$.
- Π *is a finite set of* parameter variables *over* $\mathbb{N}_0$. *Usually,* $\Pi = \{n, t, f\}$, *where* n *is the total number of processes,* t *is a bound on the number of tolerated faulty processes, and* f *is the actual number of faulty processes.*

Algorithm 1 Ben-Or's Algorithm for Byzantine Faults

```
1: bool v ← input_value({0,1})
2: int r ← 1
3: while true do
4:     send (R, r, v) to all
5:     wait for n − t messages (R, r, ∗)
6:     if received at least (n + t)/2 messages (R, r, w) then
7:         send (P, r, w, D) to all
8:     else
9:         send (P, r, ?) to all
10:    wait for n − t messages (P, r, ∗)
11:    if received at least t + 1 messages (P, r, w, D) then
12:        v ← w
13:        if received at least (n + t)/2 messages (P, r, w, D) then
14:            decide w
15:    else
16:        v ← random({0,1})
17:    r ← r + 1
```

- *RC, the* resilience condition*, is a linear integer arithmetic formula over parameter variables. E.g.: for Ben-Or,* $RC = n > 2t \land t \geq f$.
  For a vector $\mathbf{p} \in \mathbb{N}_0^{|\Pi|}$*, we write* $\mathbf{p} \models RC$ *if RC holds after substituting parameter variables with values according to* $\mathbf{p}$*. Then the set of* admissible *parameters is* $\mathbf{P}_{RC} = \{\mathbf{p} \in \mathbb{N}_0^{|\Pi|} : \mathbf{p} \models RC\}$.
- $\mathcal{R}$ *is a set of rules where a* rule *is a tuple* $r = (from, \delta_{to}, \varphi, \mathbf{uv}, \tau)$ *such that:*
  - $from \in L$ *is a location.*
  - δ_{to} *is a probability distribution over the target locations.*
  - φ *is a conjunction of lower guards and upper guards. A* lower guard *has the form:* $a_0 + \sum_{i=1}^{|\Pi|} a_i \cdot p_i \leq x$*; An* upper guard *has the form:* $a_0 + \sum_{i=1}^{|\Pi|} a_i \cdot p_i > x$*, with* $x \in \Gamma$, $a_0, \ldots, a_{|\Pi|} \in \mathbb{Q}$, $p_1, \ldots, p_{|\Pi|} \in \Pi$. *We denote these inequalities as a* lower guard on x *and an* upper guard on x *respectively.*
    The left-hand side of a lower or upper guard is called a threshold.
  - $\mathbf{uv} \in |\mathbb{N}_0|^{|\Gamma|}$ *is an* update vector *for shared variables.*
  - $\tau \subseteq \Gamma$ *is the set of shared variables to be reset to 0.*

Remark 1. For any tuple-based structure $X = (A, B, C, \ldots)$ with named components, we use dot notation to refer to individual components. For instance, $PA = (L, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$ we use $PA.L, PA.\mathcal{I}, PA.\Gamma, PA.\Pi, PA.\mathcal{R}, PA.RC$ to refer to the components $L, \mathcal{I}, \Gamma, \Pi, \mathcal{R}$, and RC respectively.

From a Probabilistic Threshold Automaton (PTA), one can derive a non-probabilistic threshold automaton simply by replacing all probabilities with non-determinism. That is, every probabilistic rule $r = (from, \delta_{to}, \varphi, \mathbf{uv}, \tau)$ is replaced by non-deterministic rules of the form $r_{to} = (from, to, \varphi, \mathbf{uv}, \tau)$ for every location

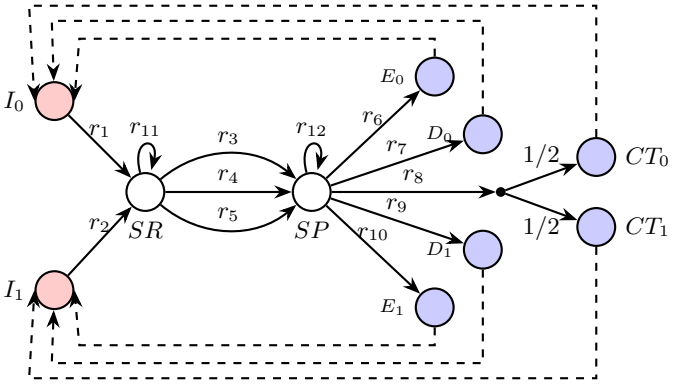

Fig. 1: Ben-Or's algorithm as a probabilistic threshold automaton with resilience condition $n > 2t \wedge t \geq f \geq 0 \wedge t > 0$. Check Table 1 for the rules' notations.

to with $\delta_{to} > 0$. While we can specify shared variables in τ to be reset, our algorithm cannot handle resets, yet.

Definition 2. *Given a PTA,* $PA = (L, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$*, its corresponding non-probabilistic threshold automaton is* $A_{np} = (L, \mathcal{I}, \Gamma, \Pi, \mathcal{R}_{np}, RC)$ *where the set of rules* $\mathcal{R}_{np}$ *is defined as follows:* $\big\{ (from, to, \varphi, \mathbf{uv}, \tau) \mid (from, \delta_{to}, \varphi, \mathbf{uv}, \tau) \in \mathcal{R} \wedge to \in L \wedge \delta_{to} > 0 \big\}$.

Example 1. Figure 2 illustrates a second, more simpler threshold automaton with the following components: $\mathcal{I} = \{V_0, V_1\}$, $L = \{V_0, V_1, \text{Wait}, D_0, D_1\}$, $\Gamma = \{x_0, x_1\}$, and $\Pi = \{n, t, f\}$. In this model, $n > 3t$ and $t \leq f$ holds. A process in V_0 has a vote of 0, while a process in V_1 has a vote of 1. The decision will be 0 (or 1) if more than $\frac{n-t}{2}$ processes vote 0 (or 1, respectively). This outcome is modeled by all processes transitioning to state D_0 (if the decision is 0) or D_1 (if the decision is 1).

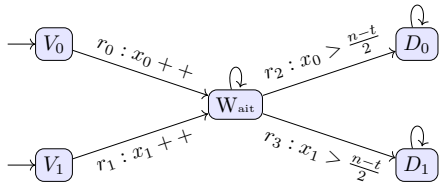

Fig. 2: A threshold automaton for simple voting.

2.1 Semantics of a Threshold Automaton

Given a TA $A = (L, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$, let the function $N : \mathbf{P}_{RC} \to \mathbb{N}_0$ determine the number of processes to be modelled, typically defined as $N(n, t, f) = n - f$, where n represents the total number of processes and f denotes the actual

Rule	Guard	Update
r_1	*true*	$x_0 ++$
r_2	*true*	$x_1 ++$
r_3	$x_0 + x_1 \geq n - t - f \ \wedge \ x_0 \geq (n+t)/2 - f$	$y_0 ++$
r_4	$x_0 + x_1 \geq n - t - f \ \wedge \ x_1 \geq (n+t)/2 - f$	$y_1 ++$
r_5	$x_0 + x_1 \geq n - t - f \ \wedge \ x_0 \geq (n-3t)/2 - f \ \wedge \ x_1 \geq (n-3t)/2 - f$	$y_? ++$
r_6	$y_0 + y_1 + y? \geq n - t - f \ \wedge \ y? \geq (n-3t)/2 - f \ \wedge \ y_0 \geq t + 1 - f$	–
r_7	$y_0 + y_1 + y? \geq n - t - f \ \wedge \ y_0 \geq (n+t)/2 - f$	–
r_8	$y_0 + y_1 + y? \geq n - t - f \ \wedge \ y? \geq (n-3t)/2 - f \ \wedge \ y? > n - 2t - f - 1$	–
r_9	$y_0 + y_1 + y? \geq n - t - f \ \wedge \ y_1 \geq (n+t)/2 - f$	–
r_{10}	$y_0 + y_1 + y? \geq n - t - f \ \wedge \ y? \geq (n-3t)/2 - f \ \wedge \ y_1 \geq t + 1 - f$	–
r_{11}	*true*	–
r_{12}	*true*	–

Table 1: Rules of the probabilistic threshold automaton for Ben-Or's algorithm in Figure 1.

number of faulty processes. The concrete semantics of a system consisting of $N(\mathbf{p})$ threshold automata running in parallel are captured by a *counter system*.

Formally, a counter system is an abstraction of $N(\mathbf{p})$ instances of a given TA running in parallel, as it only keeps track of *how many* TA are in which location, but not exactly which of the processes. However, this abstraction is well-known to be sound and complete for distributed systems with identical/anonymous processes.

Definition 3. *A* counter system *(*CS*) of a non-probabilistic threshold automaton* $A = (L, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$ *is a transition system* $\mathsf{CS}(\mathsf{A}) = (\Sigma, \Sigma_0, \mathcal{T})$ *where*

- Σ *is the set of all configurations. A* configuration *is a tuple* $\sigma = (\mathbf{k}, \mathbf{g}, \mathbf{p})$ *where:*
  - $\mathbf{k} \in \mathbb{N}_0^{|L|}$ *is a vector representing the counter values at each location. Specifically,* $\mathbf{k}[i]$ *indicates the number of processes present in location* i*. We refer to locations by their indices in* L*.*
  - $\mathbf{g} \in \mathbb{N}_0^{|\Gamma|}$ *is a vector of values for the shared variables, where* $\mathbf{g}[i]$ *is the value of variable* $x_i \in \Gamma$*.*
  - $\mathbf{p} \in \mathbf{P}_{RC}$ *is an admissible vector of parameter values.*
- *The set* Σ_0 *consists of all* initial configurations*, which satisfy the following conditions:*
  - $\forall x_i \in \Gamma : \ \sigma.\mathbf{g}[i] = 0$.
  - $\sum_{i \in \mathcal{I}} \sigma.\mathbf{k}[i] = N(\mathbf{p})$.
  - $\sum_{i \notin \mathcal{I}} \sigma.\mathbf{k}[i] = 0$.

– $\mathcal{T} \subseteq \Sigma \times \mathcal{R} \times \Sigma$ *is the set of* transitions*, where* $(\sigma, r, \sigma') \in \mathcal{T}$ *if and only if the following conditions hold:*
  - *The parameter values are unchanged:* $\sigma'.\mathbf{p} = \sigma.\mathbf{p}$.
  - *The counter value at the target location of the rule is incremented:* $\sigma'.\mathbf{k}[r.to] = \sigma.\mathbf{k}[r.to] + 1$ *(one process moves to* $r.to$*).*
  - *The counter value at the source location of the rule is decremented:* $\sigma'.\mathbf{k}[r.from] = \sigma.\mathbf{k}[r.from] - 1$ *(one process moves out of* $r.from$*).*
  - *The guard condition* $r.\varphi$ *holds in the current configuration:* $\sigma.\mathbf{g} \models r.\varphi$ *(i.e.,* φ *holds after replacing shared variables with values* $\sigma.\mathbf{g}$*).*
  - *The shared variable values are updated according to the rule:* $\sigma'.\mathbf{g} = \sigma.\mathbf{g} + r.\mathbf{uv}$.
  - *Shared variables in* τ *are reset to 0:* $\forall x_i \in \tau\ \sigma'.\mathbf{g}[i] = 0$.

  Instead of $(\sigma, r, \sigma') \in \mathcal{T}$ *we also write* $\sigma \xrightarrow{r} \sigma'$*. If* $(\sigma, r, \sigma') \in \mathcal{T}$*, we say that* r *is* enabled *in configuration* σ*; otherwise,* r *is* disabled*.*

Paths of CS. A sequence $\sigma_0, r_0, \sigma_1, \ldots, \sigma_{k-1}, r_{k-1}, \sigma_k$ of alternating configurations and rules, is called a *path* of a counter system $\mathsf{CS}(\mathsf{A}) = (\Sigma, \Sigma_0, \mathcal{T})$ if and only if the following conditions hold:

– σ_0 is an initial configuration, i.e., $\sigma_0 \in \Sigma_0$.
– For each $0 \leq i < k$, the transition $(\sigma_i, r_i, \sigma_{i+1}) \in \mathcal{T}$ holds.

In this case, we also write $\sigma_0 \rightarrow^* \sigma_k$ to denote the existence of this path. The set of all such paths in $\mathsf{CS}(\mathsf{A})$ is denoted by $Paths(\mathsf{CS}(\mathsf{A}))$.

Example 2. Let $N(n,t,f) = n - f$ and $RC = n > 3t \wedge t \geq f$. For the case where $n = 5$, $t = 1$, and $f = 1$, the following sequence represents a valid path in the counter system of the threshold automaton shown in Figure 2:
$[(4,0,0,0,0)(0,0)], \mathbf{r_0}, [(3,0,1,0,0)(1,0)], \mathbf{r_0}, [(2,0,2,0,0)(2,0)], \mathbf{r_0},$
$[(1,0,3,0,0)(3,0)], \mathbf{r_0}, [(0,0,4,0,0)(4,0)], \mathbf{r_2}, [(0,0,3,1,0)(4,0)]$.

3 Abstract Threshold Automata

A TA is an infinite state automaton due to the infinite domains of its shared variables and parameters. Therefore, to enable the use of parameterized verification techniques for finite-state processes, we introduce a finite abstract domain for the shared variables and we introduce an abstraction of TAs based on parametric interval abstraction [3].

Abstract Domain for Shared Variables. The key idea is that along a run of an automaton, we are not interested in the exact values of shared variables. Instead, we only care whether they satisfy a guard condition. Given a threshold automaton A, we define the set of thresholds as

$$\mathcal{TH} = \{d_0, d_1, \ldots, d_k\}$$

where $d_0 = 0$, $d_1 = 1$, and for all $i > 1$, d_i is a threshold in A. We assume that for all i, j, if $i < j$, then $d_i < d_j$. This ordering is always feasible for a fixed

$\mathbf{p} \in \mathbf{P}_{RC}$. If different values of $\mathbf{p} \in \mathbf{P}_{RC}$ result in different orderings of the d_i, we consider each of the finitely many such orderings separately. Based on this, we define the finite set of intervals

$$\mathcal{D} = \{I_0, I_1, \ldots, I_k\}$$

where $I_i = [d_i, d_{i+1}[$ for $i < k$, and $I_k = [d_k, \infty[$.

Definition 4. Abstract Threshold Automata [3]. *Given a threshold automaton $A = (L, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$, we define the* abstract threshold automaton *(or $\overline{TA}$) as $\overline{A} = (L, \mathcal{I}, \overline{\Gamma}, \Pi, \overline{\mathcal{R}})$, where:*

- *A and $\overline{A}$ share the components $L, \mathcal{I}, \Pi$.*
- *Let $\Gamma = \{x_0, \ldots, x_m\}$, then $\overline{\Gamma} = \{\overline{x}_0, \ldots, \overline{x}_m\}$, where each $\overline{x}_i$ takes values in the domain $\mathcal{D} = \{I_0, I_1, \ldots, I_k\}$.*
- *$\overline{\mathcal{R}}$ is the set of abstract rules. An* abstract rule *is a tuple $\overline{r} = (from, to, \overline{\varphi}, \mathbf{uv}, \tau)$, where $from, to, \mathbf{uv}, \tau$ remain unchanged from A, and the* abstract guard *$\overline{\varphi}$ is a Boolean expression over equalities between shared variables and abstract values.*
  Formally, let $\varphi = \varphi_0 \wedge \ldots \wedge \varphi_k$. Then, $\overline{\varphi} = \overline{\varphi}_0 \wedge \ldots \wedge \overline{\varphi}_k$, where:
  - *If $\varphi_i = (d_j \leq x)$, then*

$$\overline{\varphi}_i = \bigvee_{c=j}^{k-1} (\overline{x} = [d_c, d_{c+1}[) \vee (\overline{x} = [d_k, \infty[).$$

  - *If $\varphi_i = (d_j > x)$, then*

$$\overline{\varphi}_i = \bigvee_{c=0}^{j-1} (\overline{x} = [d_c, d_{c+1}[).$$

Example 3. Revisit the simple threshold automaton in Figure 2 with $N(n,t,f) = n - f$ and $RC = n > 3t \wedge t \geq f > 1$. We define $\mathcal{TH} = \{0, 1, t, \frac{n-t}{2}\}$ and $\mathcal{D} = \{[0,1[, [1,t[, [t, \frac{n-t}{2}[, [\frac{n-t}{2}, \infty[\}$, where the order is induced by the condition RC. The thresholds 0 and 1 are always included to allow detection of whether a shared variable is equal to 0.

Moreover, we have the following abstract rules:

- $\overline{r}_0 = r_0$, $\overline{r}_1 = r_1$ (due to the absence of a guard),
- $\overline{r}_2.\overline{\varphi} = (\overline{x}_0 = [\frac{n-t}{2}, \infty[)$ since the concrete guard is $x_0 > \frac{n-t}{2}$,
- $\overline{r}_3.\overline{\varphi} = (\overline{x}_1 = [\frac{n-t}{2}, \infty[)$ since the concrete guard is $x_1 > \frac{n-t}{2}$.

3.1 (0, 1)-Abstraction

Semantics of $\overline{\text{TA}}$. In this section the semantics of a system composed of an arbitrary number of $\overline{\text{TA}}$s is over-approximated by a $(0,1)$-*counter system* (or ZCS). The main component in a ZCS is the $(0,1)$-configuration. The key idea is

that, for the specifications of interest here (reachability and termination), knowing the exact number of processes at a location is unnecessary; it is sufficient to determine whether a location contains any processes. Such an abstraction transforms our system into a finite-state one, enabling the use of symbolic techniques to implement our algorithm.

A $(0,1)$-*configuration* is a tuple $\overline{\sigma} = (\overline{\mathbf{k}}, \overline{\mathbf{g}})$, where $\overline{\mathbf{k}} \in \mathbb{B}^{|L|}$ and $\overline{\mathbf{g}} \in \mathcal{D}^{|\overline{\Gamma}|}$. Here, $\overline{\mathbf{k}}[i]$ indicates the presence (1) or absence (0) of at least one process at location i, and $\overline{\mathbf{g}}$ is a vector of shared variable values, where $\overline{\mathbf{g}}[i]$ is the parametric interval currently assigned to $\overline{x}_i$. In a $(0,1)$-configuration $\overline{\sigma} = (\overline{\mathbf{k}}, \overline{\mathbf{g}})$, we denote $\overline{\mathbf{k}}$ as the 01-counter-valuation and $\overline{\mathbf{g}}$ as the 01-var-valuation. If $\overline{\mathbf{k}}[i] = 1$, we say that location i is covered in $\overline{\sigma}$.

Definition 5. *A* $(0,1)$-counter system *(or* ZCS*) [3] of an abstract threshold automaton* $\overline{A} = (L, \mathcal{I}, \overline{\Gamma}, \Pi, \overline{\mathcal{R}})$ *is a* transition system $\mathsf{ZCS}(\overline{\mathsf{A}}) = (\overline{\Sigma}, \overline{\Sigma}_0, \mathcal{T})$*, where:*

- $\overline{\Sigma} = \mathbb{B}^{|L|} \times \mathcal{D}^{|\overline{\Gamma}|}$ *is the set of* $(0,1)$*-configurations. Each configuration* $\overline{\sigma} = (\overline{\mathbf{k}}, \overline{\mathbf{g}})$ *consists of:*
  - $\overline{\mathbf{k}} \in \mathbb{B}^{|L|}$*, where* $\overline{\mathbf{k}}[i] = 1$ *indicates that location* i *contains at least one process, and* $\overline{\mathbf{k}}[i] = 0$ *otherwise.*
  - $\overline{\mathbf{g}} \in \mathcal{D}^{|\overline{\Gamma}|}$*, where* $\overline{\mathbf{g}}[i]$ *represents the parametric interval assigned to* $\overline{x}_i$*.*
- $\overline{\Sigma}_0 \subseteq \overline{\Sigma}$ *is the set of initial* $(0,1)$*-configurations satisfying:*
  - $\forall i \in \Gamma : \overline{\sigma}.\overline{\mathbf{g}}[i] = I_0$*.*
  - $\forall i \in L : \overline{\sigma}.\overline{\mathbf{k}}[i] = 1 \Leftrightarrow i \in \mathcal{I}$*.*
- *The transition relation* $\mathcal{T}$ *consists of transitions* $(\overline{\sigma}, \overline{r}, \overline{\sigma}')$*, where:*
  - $\overline{r} = \{from, to, \overline{\varphi}, \mathbf{uv}\} \in \overline{\mathcal{R}}$*.*
  - *The condition* $\overline{\sigma}.\overline{\mathbf{g}} \models \overline{r}.\overline{\varphi}$ *holds.*
  - *The source location is occupied:* $\overline{\sigma}.\overline{\mathbf{k}}[\overline{r}.from] = 1$*.*
  - *The transition updates* $\overline{\mathbf{k}}$*: either* $\overline{\sigma}'.\overline{\mathbf{k}}[\overline{r}.from] = 0$ *or it remains* 1*.*
  - *The target location is covered after the transition:* $\overline{\sigma}'.\overline{\mathbf{k}}[\overline{r}.to] = 1$*.*
  - *Shared variables are updated:* $\overline{\sigma}'.\overline{\mathbf{g}} = \overline{\sigma}.\overline{\mathbf{g}} \dot{+} \mathbf{uv}$*, defined as follows:*
    1. $\overline{\sigma}'.\overline{\mathbf{g}}[i] = \overline{\sigma}.\overline{\mathbf{g}}[i]$*, if* $\overline{r}.\mathbf{uv}[i] = 0$
    2. $(\overline{\sigma}'.\overline{\mathbf{g}}[i] = \overline{\sigma}.\overline{\mathbf{g}}[i]) \vee (\overline{\sigma}'.\overline{\mathbf{g}}[i] = \overline{\sigma}.\overline{\mathbf{g}}[i].next)$*, if* $\overline{r}.\mathbf{uv}[i] = 1$*. The first disjunct is omitted if* $\overline{\sigma}.\overline{\mathbf{g}}[i] = I_0$*.*
  - *Reset variables are reinitialized:* $\forall x_i \in \overline{r}.\tau, \overline{\sigma}'.\overline{\mathbf{g}}[i] = I_0$*.*

Paths. A *path* of $\mathsf{ZCS}(\overline{\mathsf{A}})$ is a sequence of alternating $(0,1)$-configurations and abstract rules, given by $\overline{\sigma}_0, \overline{r}_0, \overline{\sigma}_1, \ldots, \overline{\sigma}_{k-1}, \overline{r}_{k-1}, \overline{\sigma}_k$, such that for all $i < k$, the transition $(\overline{\sigma}_i, \overline{r}_i, \overline{\sigma}_{i+1})$ belongs to $\mathcal{T}$. The set of all paths of $\mathsf{ZCS}(\overline{\mathsf{A}})$ is denoted by $Paths(\mathsf{ZCS}(\overline{\mathsf{A}}))$.

Example 4. Consider again the TA from Figure 2, and $I_0 = [0,1[$, $I_1 = [1, \frac{n-t}{2}[$, $I_2 = [\frac{n-t}{2}, \infty[$. The following is a valid path of its $(0,1)$-counter system:
$[(1,0,0,0,0)(I_0,I_0)], \overline{\mathbf{r_0}}, [(1,0,1,0,0)(I_1,I_0)], \overline{\mathbf{r_0}}, [(1,0,1,0,0)(I_2,I_0)], \overline{\mathbf{r_2}}$,
$[(0,0,1,1,0)(I_2,I_0)]$.

Specifications. We say that a 01-configuration $\overline{\sigma}$ satisfies a reachability specification $L_{spec} = (L_{=0}, L_{>0})$, denoted $\overline{\sigma} \models L_{spec}$, if for all $i \in L_{=0}$, $\overline{\sigma}.\overline{\mathbf{k}}[i] = 0$, and for all $i \in L_{>0}$, $\overline{\sigma}.\overline{\mathbf{k}}[i] > 0$.

Monotonicity. Since global variables in a ZCS (or a CS) are initialized to 0 and never decrease, the following property holds:

Property 1 (Monotonicity). Given a TA A, the *monotonicity property* states that in any execution of $\mathsf{ZCS}(\overline{\mathsf{A}})$ ($\mathsf{CS}(\mathsf{A})$):

- Once a lower guard becomes enabled, it remains enabled forever.
- Once an upper guard becomes disabled, it remains disabled forever.

4 CS vs ZCS

In comparison to CS, in ZCS we drop the resilience condition, the function N that determines the number of processes to be modeled, as well as the exact number of processes in a global configuration. Moreover, a transition in ZCS may jump from one interval to the next too early and may stay in the same interval although it had to move. In our previous work [3], we showed that the abstraction from CS to ZCS is complete with respect to reachability specifications.

In the following, we show how to detect whether a behavior of the $(0, 1)$-counter system corresponds to a concrete behavior of a counter system.

A path $\overline{\pi} = \overline{\sigma}_0, \overline{r}_0, \ldots, \overline{r}_{m-1}, \overline{\sigma}_m$ in $\mathsf{ZCS}(\overline{\mathsf{A}}) = (\overline{\Sigma}, \overline{\Sigma}_0, \mathcal{T})$ *corresponds* to the paths $\pi = \sigma_0, r_0^{c_0}, \ldots, r_{m-1}^{c_{m-1}}, \sigma_m$ (where $r_i^{c_i}$ simulates applying r_i c_i times) of $\mathsf{CS}(\mathsf{A}) = (\Sigma, \Sigma_0, \mathcal{T})$ that satisfy the following conditions:

- $RC \wedge (\sum_{j \in \mathcal{I}} \sigma_0.\mathbf{k}[j] = N(n, t, f))$
- $\forall i < m\ \sigma_i.\mathbf{k}[r_i.from] = c_i + \sigma_{i+1}.\mathbf{k}[r_i.from] \wedge \sigma_{i+1}.\mathbf{k}[r_i.to] = c_i + \sigma_i.\mathbf{k}[r_i.to]$
- $\forall i < m\ \forall x_j \in \Gamma\ x_j \notin r_i.\tau \implies \sigma_{i+1}.\mathbf{g}[j] = \sigma_i.\mathbf{g}[j] + c_i \cdot r_i.\mathbf{uv}[j]$
- $\forall i < m\ \forall x_j \in r_i.\tau\ \sigma_{i+1}.\mathbf{g}[j] = 0$
- $\forall i < m\ \forall x_j \in \Gamma\ \sigma_i.\mathbf{g}[j] \in \overline{\sigma}_i.\overline{\mathbf{g}}[j] \wedge \sigma_{i+1}.\mathbf{g}[j] \in \overline{\sigma}_{i+1}.\overline{\mathbf{g}}[j]$
- $\forall i < m\ c_i > 1 \implies ((\sigma_{i+1}.\mathbf{g} - r_i.\mathbf{uv}) \models r_i.\varphi)$[1]

Let $Concretize(\overline{\pi})$ denote the conjunction of the constraints above, where quantified formulas are instantiated as a finite conjunction of quantifier-free formulas. Since $Concretize(\overline{\pi})$ is a quantifier-free formula in linear integer arithmetic, a satisfying assignment (which can be computed by an SMT solver) represents a path of $\mathsf{CS}(\mathsf{A})$ corresponding to $\overline{\pi}$. A path $\overline{\pi} \in Paths(\mathsf{ZCS}(\overline{\mathsf{A}}))$ is said to be *spurious* if $Concretize(\overline{\pi})$ is unsatisfiable.

5 Almost-Sure Termination and SCC Detection

In our previous work [3], we introduced a reachability algorithm for ZCS. The algorithm begins with the set of target states and performs a backward traversal

[1] This is needed only in cases where an update affects any of the guards of $r_i.\varphi$

of the state space until it reaches a fixed point. Within this fixed point, all paths originating from the initial state are examined to determine whether they are spurious, as described in Section 4. If at least one such path is not spurious, the algorithm concludes that the target states are reachable. This reachability algorithm serves as two subprocedures, *IsReachable* and *ComputeFixedPoint*, in Algorithm 2.

In this section, we present our algorithm (Algorithm 2) for proving almost-sure termination and for detecting strongly connected components (SCCs) in probabilistic threshold automata, as well as for proving termination and detecting SCCs in ordinary threshold automata. Then we prove that our algorithm is sound. We assume that the input probabilistic automaton is deterministic, reset-free, and that its corresponding $(0, 1)$-counter system is free of both local and global deadlocks.

Remark 2. A fault-tolerant system terminates if and only if all correct processes (modeled by a TA) reach a final state.

Before presenting the algorithm, we first provide the following definitions of the key terms and concepts used throughout the algorithm.

- For a given set S, let $\mathcal{P}_\emptyset(S)$ denote the powerset of S excluding the empty set.
- A set $X \subset V$ is strongly connected if, for every two elements $v, u \in X$, there is a path from v to u. A *strongly connected component* (SCC) is a maximal strongly connected set $S \subseteq V$. We denote a strongly connected component over L by local SCC, and a strongly connected component over $\overline{\Sigma}$ by simply SCC.
- A *strongly connected component* (SCC) of a directed graph is a maximal subgraph where for any two vertices u and v, there exists a path from u to v and vice versa. We refer to an SCC within a TA as a *local SCC* and an SCC within ZCS as an 01-*SCC*. Given a local SCC $\mathcal{C}$, we denote by $\mathcal{C}.Locations$ the set of locations in $\mathcal{C}$, and by $\mathcal{C}.Rules$ the set of rules in $\mathcal{C}$. We use the same notations for a 01-SCC.
- A local SCC $\mathcal{C}$ is *valid* if there exists at least one $(0, 1)$-*configuration* $\overline{\sigma} = (\overline{\mathbf{k}}, \overline{\mathbf{g}})$ where $\forall \overline{r}_i \in \mathcal{C}.Rules : \overline{\mathbf{g}} \models \overline{r}_i.\overline{\varphi}$. Otherwise, $\mathcal{C}$ is called *invalid*.
- A local SCC $\mathcal{C}$ is *finite-traverse* if there exist indices i, j and a variable k such that $\overline{r}_i, \overline{r}_j \in \mathcal{C}.Rules$ where $\overline{r}_i.\mathbf{uv}[k] > 0$ and $\overline{r}_j.\overline{\varphi}$ includes an upper guard on k. Otherwise, $\mathcal{C}$ is *infinite-traverse*. A local SCC $\mathcal{C}$ is valid-infinite if $\mathcal{C}$ is valid and infinite-traverse.
- We extend the latter definition to subsets of local SCCs as follows: A *finite-traverse subset* is a subset of local SCCs $S^{\mathcal{C}} = \{\mathcal{C}_1, \ldots, \mathcal{C}_c\}$ for which there exist indices i, j, rules $\overline{r}_i, \overline{r}_j$, and a variable k such that $\mathcal{C}_i, \mathcal{C}_j \in S^{\mathcal{C}}$ where $\overline{r}_i \in \mathcal{C}_i.Rules \wedge \overline{r}_i.\mathbf{uv}[k] > 0$, and $\overline{r}_j \in \mathcal{C}_j.Rules$ with $\overline{r}_j.\overline{\varphi}$ includes an upper guard on k. Otherwise, $S^{\mathcal{C}}$ is called an *infinite-traverse subset*.

Algorithm 2 Termination and SCC detection. If the given automaton is probabilistic, it is converted first to its non-deterministic version (see Definition 2).

Input: Abstract TA ($\overline{\text{TA}}$), final locations, and a probabilistic flag
Output: termination flag.
procedure TerminationCheck($ATA, final\_locs, isProbabilistic$)
    $localSCCs \leftarrow ExtractAllSCCs(ata)$ // Extract all local SCCs from the *ata*.
    // keep only infinite-traverse local SCCs.
    $localSCCs \leftarrow \{\mathcal{C} \in localSCCs \mid \mathcal{C}.IsInfiniteTraverse\}$
    $sccValidIntrvls \leftarrow \emptyset$
    // keep only valid local SCCs.
    for all $\mathcal{C} \in localSCCs$ **do**
        // *varGuards* maps a variable to its guards on the local SCC.
        $varGuards \leftarrow GetVarGuards(\mathcal{C})$
        for all $(var, guards) \in varGuards$ **do**
            // *varValidIntrvls* maps a var to intervals satisfying *guards*.
            $varValidIntrvls \leftarrow GetVarEnablingIntrvls(var, guards)$
            if $\exists var \in varValidIntrvls.Keys : varValidIntrvls[var] == \emptyset$ **then**
                $localSCCs \leftarrow localSCCs \setminus \{\mathcal{C}\}$ // remove invalid local SCCs
            else
                $sccValidIntrvls[\mathcal{C}] \leftarrow \bigotimes_{V \in \text{varValidIntrvls.Values}} V$
                // $sccValidIntrvls[\mathcal{C}] = \{\overline{\mathbf{g}} \mid \forall \overline{r} \in \mathcal{C}.Rules : \overline{\mathbf{g}} \models \overline{r}.\overline{\varphi}\}$
    $subsets \leftarrow \mathcal{P}_\emptyset(localSCCs)$ // generate all subsets.
    // keep only infinite-traverse subsets.
    $subsets \leftarrow \{subset \in subsets \mid subset.isInfiniteTraverse\}$
    if $isProbabilistic$ **then**
        // 01-configurations in which only final locations are covered.
        $final01Configs \leftarrow ConvertTo01Configs(\mathcal{P}_\emptyset(final\_locs))$
    $scc01Configs \leftarrow \emptyset$
    for all $subset \in subsets$ **do**
        // $subsetValidIntrvls[subset] = \{\overline{\mathbf{g}} \mid \forall \overline{r} \in subset : \overline{\mathbf{g}} \models \overline{r}.\overline{\varphi}\}$
        $subsetValidIntrvls[subset] \leftarrow \bigcap_{\mathcal{C} \in subset} sccValidIntrvls[\mathcal{C}]$
        if $subsetValidIntrvls[subset] == \emptyset$ **then** *continue*
        $locsSubsets \leftarrow \mathcal{P}_\emptyset(subset.Locations)$
        // convert 01-counter-valuations and 01-var-valuations into 01-configuration
        $01Configs \leftarrow locsSubsets \otimes subsetValidIntrvls[subset]$
        // check if traversing *subset* infinitly has probability 0.
        if $isProbabilistic$ **then**
            $nonDetImage \leftarrow ComputeNonDetImage(01Configs)$
            $fixedPoint \leftarrow ComputeFixedPoint(final01Configs)$
            if $fixedPoint \cap nonDetImage \neq \emptyset$ **then** $01Configs \leftarrow \emptyset$
        // collcet all configurations that are on an SCC
        $scc01Configs \leftarrow scc01Configs \cup 01Configs$
    Return $IsReachable(scc01Configs)$

5.1 Detailed Description for Algorithm 2

Algorithm 2 starts in Line 4 by extracting all local SCCs ($localSCCs$) in $\overline{\text{TA}}$. This can be computed efficiently using either Tarjan's algorithm, which runs in linear time, or the symbolic CHAIN algorithm [15], which is also linear. Line 6 removes all *finite-traverse* local SCCs from $localSCCs$. Lines 9 to 19 filter out invalid local SCCs and compute, for each *valid infinite-traverse* local SCC, the set of 01-var-valuations that enable all its rules. The cross-product operation in Line 18 can be efficiently implemented using a BDD manager by computing the conjunction of disjunctions over the intervals of each set in $varValidIntrvls.Values$. At this stage all local SCCs in $localSCCs$ are guaranteed to be valid-infinite due to Monotonicity property (see Property 1). Lines 20 to 22 compute all subsets of *infinite-traverse* local SCCs. Line 25 computes all $(0,1)$-configurations in which only final locations ($final\_locs$) are covered. Line 29 computes the set of 01-var-valuations that enable all rules in every cycle of a given *subset*. If and only if *subset* is valid (Line 30), its corresponding 01-counter-valuations and 01-var-valuations are converted into $(0,1)$-configurations (Lines 31-33). Lines 35 to 38 check whether traversing the local SCC set (*subset*) infinitely has probability 0. The key idea is that if infinite travese local SCC $\mathcal{C} \in subset$ contains a non-deterministic transition, one remaining within $\mathcal{C}$ and another leading outside and reaches a final location, then the probability of infinitely choosing the intra-SCC transition is 0. $ComputeNonDetImage(01Configs)$ in Line 36 is a procedure that computes the image of $01Configs$ as follows:

- Select all $(0,1)$-configurations σ from $01Configs$ that satisfy the following condition: If a location l belongs to $locsSubsets$ and is covered in σ, then l has a non-deterministic transition to a location outside $locsSubsets$. We refer to this set as NDT.
- For each $(0,1)$-configuration $\sigma \in NDT$, compute a new $(0,1)$-configuration σ'. The configuration σ' is obtained by executing all non-deterministic transitions that originate from locations in $locsSubsets$ (covered in σ) and lead to locations outside $locsSubsets$.

Line 40, collects all $(0,1)$-configurations that belong to a 01-SCC, and checks if any configuration in $scc01Configs$ is reachable. The procedures in Lines 37 and 41 follow the approach we presented in [3] which also takes care of spurious paths.

Remark 3. To simplify the presentation of the algorithm, we assumed that a local SCC is either *valid-infinite* or not. However, in general, a local SCC that is not *valid-infinite* can be decomposed into multiple *valid-infinite* local SCCs.

5.2 Correctness

Algorithm 2 is sound. Soundness follows directly from the fact that $scc01Configs$ contains all $(0,1)$-configurations that may traverse a 01-SCC inifintely often. Also

since, for probabilistic systems, SCCs with a probability of 0 of being traversed infinitely often are excluded (see Lines 35 - 38),

Theorem 1 (Soundness SCC Detection). *Algorithm 2 is sound for SCC detection. That is, if the algorithm computes a non-empty* $scc01Configs$*, then every* $(0,1) - configuration$ *in* $scc01Configs$ *belongs to a* 01*-SCC.*

The below corollary is obtained since SCC detection is sound and since the reachability algorithm in [3] is proven to be correct.

Theorem 2 (Soundness (A.S.) Termination). *Algorithm 2 is sound for termination and almost-sure termination. Specifically, if the algorithm returns true, it ensures that a system consisting of a network of TAs does not terminate, regardless of its size.*

We will leave completness and the handling of resets for future work.

6 Related Work

Recently, there have been many works [2,3,12–14] that target the parameteried verification of non-probabilistic threshold automata. In [12,13], Konnov et al. proposed an approach for detecting TA traces that violate reachability specifications, as well as lasso-shaped TA traces that violate a given liveness property [13]. These methods have been implemented in the ByMC tool [14]. The decidability and complexity of verification and synthesis for threshold automata were also addressed in [2]. Their decision procedure relies on an SMT encoding of potential error paths, where, in general, the size of the SMT formula increases exponentially with the length of the paths. In [3], the authors extended the threshold automaton with resets and variable decrements, and introduced a new algorithm to check reachability and coverability. Additionally, their approach removed the cycle absence restriction that was required in previous works.

In [6], the authors introduced the probabilistic threshold automaton, a threshold automaton extended with coin tosses, and proposed a new approach for verifying them and checking almost-sure termination. This approach works under two key restrictions, which we lift in this paper: 1) it does not allow cycles inside the automaton, and 2) coin tosses may only appear at the end of a round. An extension of the PTA was presented in [8], where the authors incorporated common coins into PTAs. They reduced the formal verification of the extended PTA to single-round queries on non-probabilistic threshold automata, which are then verified using ByMC [12].

Unfortunately, there are few works [5,7,16,17,20,24] that address the automatic verification of probabilistic parameterized systems. Unlike this work (and [6,8]), these approaches rely on process templates with a finite state space and use a single parameter, the number of process template instances. In [5], the authors use a probabilistic single-clock timed automaton with a broadcasting communication primitive. They verify whether a configuration in which one

process reaches a target state almost surely. Their approach is based on well-structured transition systems. In [17], a method was introduced to prove liveness for randomized parameterized systems under arbitrary schedulers, while [16] presented a fully automatic verification method for proving almost-sure termination of probabilistic parameterized concurrent systems. Both approaches [5,16,17] are based on regular model checking and do not support arithmetic resilience conditions or shared variables over infinite domains. The seminal work by Pnueli and Zuck [20] requires shared variables to be over finite domains and restricts the use of thresholds to only 1 and n. In [24], authors presented a novel approach for checking liveness in probabilistic parameterized protocols, by abstracting a parameterized Markov Decision Process (MDP) to a finite MDP.

7 Conclusion

In this paper, we introduced an extension of probabilistic threshold automata to support modelling resets of shared variables. We further presented a sound algorithm for detecting almost-sure termination in randomized fault-tolerant algorithms, modelled with the help of probabilistic threshold automata, and for verifying termination in ordinary fault-tolerant algorithms, albeit so far only for reset-free automata. Our approach enables the detection of strongly connected components and termination in system models consisting of an arbitrary number of threshold automata. Additionally, it identifies strongly connected components and verifies almost-sure termination in system models with an arbitrary number of probabilistic threshold automata. Furthermore, we lifted two key restrictions previously imposed on threshold automata: (1) cycles were not allowed, and (2) coin tosses could only appear at the end of the automaton. For future work, we plan to investigate the completeness of our approach, handle resets of shared variables, relax the deadlock restriction by incorporating a fairness notion, and develop an implementation of the proposed algorithm.

References

1. Baier, C., Daum, M., Engel, B., Härtig, H., Klein, J., Klüppelholz, S., Märcker, S., Tews, H., Völp, M.: Chiefly symmetric: Results on the scalability of probabilistic model checking for operating-system code. In: Prof. of the 7th Conference on Systems Software Verification (SSV'12). Electronic Proceedings in Theoretical Computer Science, vol. 102, pp. 156–166 (2012). https://doi.org/10.4204/EPTCS.102.14
2. Balasubramanian, A., Esparza, J., Lazić, M.: Complexity of verification and synthesis of threshold automata. In: International Symposium on Automated Technology for Verification and Analysis. pp. 144–160. Springer (2020)
3. Baumeister, T., Eichler, P., Jacobs, S., Sakr, M., Völp, M.: Parameterized verification of round-based distributed algorithms via extended threshold automata. In: International Symposium on Formal Methods. pp. 638–657. Springer (2024)
4. Ben-Or, M.: Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In: Proceedings of the Second Annual ACM

Symposium on Principles of Distributed Computing. p. 27–30. PODC '83, Association for Computing Machinery, New York, NY, USA (1983). https://doi.org/10.1145/800221.806707, https://doi.org/10.1145/800221.806707
5. Bertrand, N., Fournier, P.: Parameterized verification of many identical probabilistic timed processes. In: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2013). pp. 501–513. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2013)
6. Bertrand, N., Konnov, I., Lazić, M., Widder, J.: Verification of randomized consensus algorithms under round-rigid adversaries. International Journal on Software Tools for Technology Transfer **23**(5), 797–821 (2021)
7. Esparza, J., Gaiser, A., Kiefer, S.: Proving termination of probabilistic programs using patterns. In: Computer Aided Verification: 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings 24. pp. 123–138. Springer (2012)
8. Gao, S., Zhan, B., Wu, Z., Zhang, L.: Verifying randomized consensus protocols with common coins. In: 2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 403–415 (2024). https://doi.org/10.1109/DSN58291.2024.00047
9. Guerraoui, R., Kuznetsov, P., Monti, M., Pavlovic, M., Seredinschi, D.: Scalable byzantine reliable broadcast. In: Suomela, J. (ed.) 33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary. LIPIcs, vol. 146, pp. 22:1–22:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPICS.DISC.2019.22, https://doi.org/10.4230/LIPIcs.DISC.2019.22
10. Jacobs, S., Sakr, M.: Analyzing guarded protocols: Better cutoffs, more systems, more expressivity. In: International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 247–268. Springer (2018)
11. Jacobs, S., Sakr, M., Zimmermann, M.: Promptness and bounded fairness in concurrent and parameterized systems. In: International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 337–359. Springer (2020)
12. Konnov, I., Lazić, M., Veith, H., Widder, J.: Para 2: parameterized path reduction, acceleration, and smt for reachability in threshold-guarded distributed algorithms. Formal Methods in System Design **51**(2), 270–307 (2017)
13. Konnov, I., Lazić, M., Veith, H., Widder, J.: A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 719–734 (2017)
14. Konnov, I., Widder, J.: Bymc: Byzantine model checker. In: International Symposium on Leveraging Applications of Formal Methods. pp. 327–342. Springer (2018)
15. Larsen, C.A., Schmidt, S.M., Steensgaard, J., Jakobsen, A.B., de Pol, J.v., Pavlogiannis, A.: A truly symbolic linear-time algorithm for scc decomposition. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 353–371. Springer (2023)
16. Lengál, O., Lin, A.W., Majumdar, R., Rümmer, P.: Fair termination for parameterized probabilistic concurrent systems. In: Tools and Algorithms for the Construction and Analysis of Systems: 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I 23. pp. 499–517. Springer (2017)

17. Lin, A.W., Rümmer, P.: Liveness of randomised parameterised systems under arbitrary schedulers. In: International Conference on Computer Aided Verification. pp. 112–133. Springer (2016)
18. Malkhi, D., Reiter, M.: Byzantine quorum systems. In: Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing. p. 569–578. STOC '97, Association for Computing Machinery, New York, NY, USA (1997). https://doi.org/10.1145/258533.258650, https://doi.org/10.1145/258533.258650
19. Neiheiser, R., Matos, M., Rodrigues, L.: Kauri: Scalable bft consensus with pipelined tree-based dissemination and aggregation. In: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. pp. 35–48 (2021)
20. Pnueli, A., Zuck, L.: Verification of multiprocess probabilistic protocols. In: Proceedings of the third annual ACM symposium on Principles of distributed computing. pp. 12–27 (1984)
21. Suzuki, I.: Proving properties of a ring of finite-state machines. Inf. Process. Lett. **28**(4), 213–214 (1988). https://doi.org/10.1016/0020-0190(88)90211-6, https://doi.org/10.1016/0020-0190(88)90211-6
22. Zamani, M., Movahedi, M., Raykova, M.: Rapidchain: Scaling blockchain via full sharding. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 931–948. ACM (2018). https://doi.org/10.1145/3243734.3243853, https://doi.org/10.1145/3243734.3243853
23. Zarbafian, P., Gramoli, V.: Lyra: Fast and scalable resilience to reordering attacks in blockchains. In: 2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 929–939. IEEE (2023)
24. Zuck, L.D., McMillan, K.L., Torf, J.: : Planner-less proofs of probabilistic parameterized protocols. In: International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 336–357. Springer (2017)