# Holistic Verification of Blockchain Consensus

NATHALIE BERTRAND, Univ Rennes, Inria, CNRS, IRISA, France
VINCENT GRAMOLI, University of Sydney, Australia
IGOR KONNOV, Informal Systems, Austria
MARIJANA LAZIĆ, TU Munich, Germany
PIERRE THOLONIAT, Columbia University, USA
JOSEF WIDDER, Informal Systems, Austria

Blockchain has recently attracted the attention of the industry due, in part, to its ability to automate asset transfers. It requires distributed participants to reach a consensus on a block despite the presence of malicious (a.k.a. Byzantine) participants. Malicious participants exploit regularly weaknesses of these blockchain consensus algorithms, with sometimes devastating consequences. In fact, these weaknesses are quite common and are well illustrated by the flaws in the hand-written proofs of existing blockchain consensus protocols [63]. Paradoxically, until now, no blockchain consensus has been holistically verified using model checking.

In this paper, we remedy this paradox by model checking for the first time a blockchain consensus used in industry. We propose a holistic approach to verify the consensus algorithm of the Red Belly Blockchain [20], for any number $n$ of processes and any number $f < n/3$ of Byzantine processes. We decompose directly the algorithm pseudocode in two parts—an inner broadcast algorithm and an outer decision algorithm—each modelled as a threshold automaton [36], and we formalize their expected properties in linear-time temporal logic. We then automatically check the inner broadcasting algorithm, under a carefully identified fairness assumption. For the verification of the outer algorithm, we simplify the model of the inner algorithm by relying on its checked properties. Doing so, we formally verify not only the safety properties of the Red Belly Blockchain consensus but also its liveness in about 70 seconds.

## 1 INTRODUCTION

### 1.1 Context

Today, the market capitalization of the seminal blockchain, called Bitcoin, is about \$803B[1], which incentivizes malicious participants to find problematic executions that would allow them to steal financial assets. As the blockchain requires a distributed set of machines to agree on a unique block of transactions to be appended to the chain, attackers naturally try to exploit consensus vulnerabilities: they force participants to disagree so that they wrongly believe that two conflicting transactions are legitimate, leading to what is known as a *double spending*. In 2014, malicious participants managed to exploit Bitcoin consensus vulnerabilities to steal \$83,000 through a network attack. In August 2021, 570,000 transactions were reverted in a more recent version of Bitcoin, Bitcoin SV, by forcing its blockchain consensus protocol to violate its safety property (i.e., agreement). With 3 attacks on the same blockchain within 4 months, thefts are becoming commonplace.[2] Unsurprisingly, various bugs in specifications and in proofs of blockchain consensus protocols appear in the literature [1, 62]. This is illustrated by the flaw in the consensus algorithms now used in in-production blockchains [63]. The crux of the problem is that reasoning about distributed executions of blockchain consensus protocols is hard due to several sources of non-determinism, and in particular asynchrony and faults. As a result, formally verifying that a blockchain consensus protocol is safe and live is key to mitigate financial losses.

Recent progress in mechanical proofs represent the first steps towards verifying blockchain consensus. For instance, parameterized model checking aims at verifying algorithms for an arbitrary number $n$ of processes [11] that is unknown at design time. In some contexts, it reduces the model checking for any fault number $f$ and its upper bound $t$ to bounded

[1] https://coinmarketcap.com.
[2] cointelegraph.com/news/bitcoin-sv-rocked-by-three-51-attacks-in-as-many-months

model checking questions [29]. The threshold automaton (TA) framework for communication-closed algorithms [7, 36] targets algorithms with thresholds in guards such as "number of messages from distinct processes exceeds $2t + 1$", and in the resilience condition, typically of the form $n > 3t$. The parameterized model checking of threshold automata builds upon a reduction [26, 42] that reorders steps of asynchronous executions to obtain simpler executions, which are equivalent to the original executions with respect to safety and liveness properties. Such a technique has recently proved instrumental in verifying fully asynchronous parts of consensus algorithms, like broadcast algorithms [36].

Due to the famous unfeasability of deterministic consensus in asynchronous setting [28], this promising method was not applied to proving deterministic consensus algorithms[3]. In fact, the aforementioned reduction technique cannot apply to partial synchrony [25]: moving the message reception step to a later point in the execution might violate an assumed message delay. Yet, these delays are important as typical partially synchronous consensus algorithms feature timers to catch up with the unknown bound on the delay to receive a message. Most known verification techniques therefore target either synchronous (lock-step) or asynchronous semantics. In addition, partially synchronous consensus algorithms generally rely on a coordinator process that helps other processes converge and whose identifier rotates across rounds. Some efforts have been devoted to proving the termination of partially synchronous consensus algorithms, like Paxos, assuming synchrony [30]. The drawback is that such algorithms aim at tolerating non-synchronous periods before reaching a global stabilization time (GST) after which they terminate. Proving that such an algorithm terminates under synchrony does not show that the algorithm would also terminate if processes reached GST at different points of their execution. Instead, one would also need to show that correct processes can catch up in the same round. This would, in turn, require proving the correctness of a synchronizer algorithm [25].

Verifying consensus is even more subtle when processes are Byzantine as they can execute arbitrary steps, changing their local state and the values they share. One needs to reason about executions with all possible scenarios resulting from arbitrary behaviors, multiplying the already large number of interleaved executions. The verification of such algorithms is thus either restricted to showing safety properties, like agreement and validity, and ignoring liveness [40]; or to proving separate parts of the blockchain consensus [44]. Such noticeable efforts are well illustrated with the series of attempts to verify the Stellar Consensus Protocol (SCP) [43]: Ivy is used to model a slightly different algorithm in which the key novelty of the SCP consensus algorithm, which is its dynamic quorum system [51] called quorum slices, is replaced by static quorums [45]. In addition, it relies on axioms some of which are proved separately in Isabelle/HOL without being linked to the Ivy axioms [22]. Without a holistic approach, the verification of the components of a protocol does not imply that the protocol is verified.

### 1.2 Contributions

In this paper, we verify holistically the safety and liveness properties of a Byzantine consensus used in the Red Belly Blockchain system [20], a scalable blockchain used in production. Our approach is holistic because it starts from the pseudocode of the distributed algorithm as typically presented in the distributed computing literature, models this pseudocode and its components into disambiguated threshold automata (TAs), model checks the desired properties of these components expressed in LTL formulae, simplifies the TA of the consensus algorithm with these verified properties and model checks the safety and liveness of the consensus protocol. The advantage is that the formally verified algorithm matches the pseudocode and no user-defined invariants or proofs need to be checked, which drastically reduces the risks of human errors.

[3] Here deterministic means that randomization is forbidden. However, the environment (*e.g.*, communication delays, scheduler) introduces non-determinism in the algorithm execution.

(1) We formally verify a Byzantine consensus algorithm [19] used for e-voting [14], accountability [18] and blockchain [20]. This consensus algorithm now runs in the network of the Red Belly Blockchain [20] maintained by the Red Belly Network company. It executes in asynchronous rounds that broadcast binary values and compare the delivered values to the parity of the round to decide. To model check the algorithm holistically, we replace the partial synchrony assumption by a fairness assumption. Interestingly, our fairness assumption only requires that in any infinite sequence of rounds, there exists a round where, at all correct processes, a broadcast instance delivers the same binary value, or bit, first.

(2) We exploit the modularity of distributed algorithms in parameterized model checking. We first model the consensus algorithm into two simpler algorithms modeled as threshold automata (TAs): (i) an inner broadcast TA modeling a binary value variant of the reliable broadcast [47] and (ii) an outer decision TA modeling a round-based execution that inspects the delivered messages [19] to decide. We express the guarantees of the inner broadcast primitive as temporal logic properties that we automatically verify and we replace the inner TA in the global TA by a gadget TA that captures the proven temporal specification. We automatically verify the global TA with model checking.

(3) We show the practicality of our verification technique by running the parameterized model checker ByMC [36] for any number $n$ of processes and any arbitrary number $f < n/3$ of Byzantine processes. We compare the execution times when model checking the naive TA encoding the consensus algorithm and when model checking both the inner TA encoding the broadcast algorithm and then the outer TA. We demonstrate empirically that, although a parallel execution of ByMC on 64 cores could not prove the safety of the naive TA within 3 days, it proves both the liveness and safety of the simplified TA in about 70 seconds.

### 1.3 Outline

In Section 2 we introduce our preliminary definitions, in Section 3 we model our binary value broadcast algorithm pseudocode into a corresponding threshold automaton, in Section 4 we explain how the formal verification of the properties of the broadcast algorithm helps us model check the consensus algorithm and in Section 5 we verify the consensus algorithm. In Section 6 we present the results of the model checker. In Section 7, we present the related work and in Section 8, we conclude. In the Appendix we explain the multiple-round TA to one-round TA reduction (A), provide examples related to fairness (B), missing proofs (C and E) and detailed specifications (D and F).

## 2 PRELIMINARIES

The consensus algorithm runs over $n$ asynchronous sequential processes from the set $\Pi = \{p_1, \dots, p_n\}$. The processes communicate by exchanging messages through an asynchronous reliable fully connected point-to-point network, hence there is no bound on the delay to transfer a message but this delay is finite.

**Failure model.** Up to $t < n/3$ processes can exhibit a *Byzantine* behavior [54], and behave arbitrarily. We refer to $f \leq t$ as the actual number of Byzantine processes. A Byzantine process is called *faulty*, a non-faulty process is *correct*.

**Algorithm semantics.** The asynchronous semantics of a distributed algorithm executed by processes in $\Pi$ assumes discrete time and at each point in time, exactly one process takes a step. We assume that two messages cannot be received at the same time by the same process. The global execution then consists in an interleaving of the individual steps taken by the processes. Process $p_i$ sends a message to $p_j$ by invoking the primitive "send $\textsc{header}(m)$ to $p_j$", where

HEADER indicates the type of message and $m$ its contents. Process $p_i$ receives a message by executing the primitive "receive()". The shorthand broadcast(HEADER, $m$) represents "for each $p_j \in \Pi$ do send HEADER($m$) to $p_j$". And the right arrow in broadcast(HEADER, $m$) $\longrightarrow$ *messages* indicates, when specified, that "upon reception of HEADER($m$) from process $p'_j$ do $messages[p'_j] \leftarrow messages[p'_j] \cup \{m\}$". The process id is used as a subscript to denote that a variable is local to a process—for instance $var_i$ is local to process $p_i$—and is omitted when it is clear from the context.

The verification method considered in this paper exploits the fact that the algorithms are communication-closed [26], *i.e.* only messages from the current loop iteration or *round* of a process may influence its steps. This can be implemented by tagging every message by its round number $r$; during round $r$ all received messages with tag $r' < r$ are discarded and all received messages with tag $r' > r$ are stored for later.

**The consensus problem.** Assuming that each correct process proposes a binary value, the binary Byzantine consensus problem is for each of them to decide on a binary value in such a way that the following properties are satisfied:

(1) Termination. Every correct process eventually decides on a value.
(2) Agreement. No two correct processes decide on different values.
(3) Validity. If all correct processes propose the same value, no other value can be decided.

**Threshold automaton (TA).** A *threshold automaton* [37] describes the behavior of a process in a distributed algorithm. Its nodes are *locations* representing local states, and labeled edges are *guarded rules*. Formally, it is a tuple $\langle \mathcal{L}, \mathcal{I}, \Gamma, \mathcal{P}, \mathcal{R}, RC \rangle$ where $\mathcal{L}$ is the set of locations, $\mathcal{I} \subset \mathcal{L}$ is the set of initial locations, $\Gamma$ is the set of shared variables that all processes can update, $\mathcal{P}$ is the finite set of parameter variables, $\mathcal{R}$ is the set of rules, and $RC$ is the resilience condition over $\mathbb{N}_0^{|\Pi|}$. Rules are defined as tuples $\langle \mathit{from}, \mathit{to}, \phi, \vec{u} \rangle$, where *from* (resp. *to*) describes the source (resp. destination) locations, and the rule label is $\phi \mapsto \vec{u}$. Formula $\phi$ is called a *threshold guard* or simply a *guard*.

```
bv-broadcast(BV, $\langle val, i \rangle$):
  broadcast(BV, $\langle val, i \rangle$)
  **repeat:**
    **if** (BV, $\langle v, * \rangle$) received from $(t+1)$ distinct processes but not yet re-broadcast **then**
      broadcast(BV, $\langle v, i \rangle$)
    **if** (BV, $\langle v, * \rangle$) received from $(2t+1)$ distinct processes **then**
      $contestants \longleftarrow contestants \cup \{v\}$
```

Fig. 1. The pseudocode of the binary value broadcast for process $p_i$.

*Example 2.1.* As an example, Fig. 1 presents the pseudocode of the binary value broadcast and Fig. 2 its TA. (The modeling of pseudocode (Fig. 1) into TA (Fig. 2) will be described in detail in Section 3.1.) To illustrate the TA notations, note that two of the locations in $\mathcal{L} = \{V_0, V_1, B_0, B_1, B_{01}, C_0, C_1, CB_0, CB_1, C_{01}\}$ are initial: $\mathcal{I} = \{V_0, V_1\}$. Shared variables are $b_0$ and $b_1$ and can be updated by each process traversing the TA, while parameters are $n$, $t$ and $f$ and remain unchanged across the execution. The set of rules $\mathcal{R}$ consists of $\{r_i \mid 1 \leq i \leq 12\}$ together with 7 self-loops. The self-loops mimic the asynchrony between processes in the system. For example, rule $r_3$ is defined as $\langle B_0, C_0, b_0 \geq 2t+1-f, \vec{0} \rangle$. The resilience condition is $n > 3t \land t \geq f \geq 0$.

A *multi-round threshold automaton* is intuitively defined such that one round is represented by a threshold automaton, and additional so-called *round-switch rules* connect final locations with initial ones, and therefore allow processes to

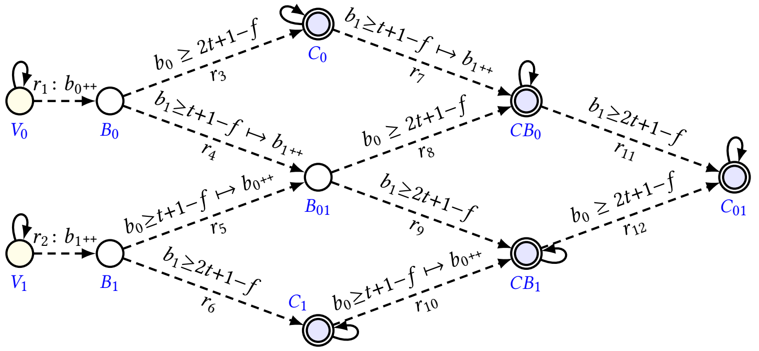

Fig. 2. The threshold automaton model for the binary value broadcast.

move from one round to the following one. We typically depict those round-switch rules as dotted arrows. Examples of such multi-round TA are depicted later in Figures 3 and 4. When it is clear from the context that there are multiple rounds, we simply call them threshold automata, and to stress that a TA does not have multiple rounds, we may call it a one-round TA.

**Counter systems.** The semantics of a (one-round) threshold automaton TA are given by a counter system $Sys(\mathsf{TA}) = \langle \Sigma, I, T \rangle$ where $\Sigma$ is the set of all configurations among which $I$ are the initial ones, and $T$ is the transition relation. A configuration $\sigma \in \Sigma$ of a one-round TA captures the values of location counters (counting the number of processes at each location of $\mathcal{L}$, therefore non-negative integers), values of global variables, and parameter values. A transition $t \in T$ is *unlocked in $\sigma$* if there exists a rule $r = \langle from, to, \phi, \vec{u} \rangle \in \mathcal{R}$ such that $\phi$ evaluates to true in $\sigma$, and location counter of *from* is at least 1, denoted $\kappa[from] \geq 1$, showing that at least one process is currently in *from*. In this case we can execute transition $t$ on $\sigma$ by moving a process along the rule $r$ from location *from* to location *to*, which is modeled by decrementing counter $\kappa[from]$, incrementing $\kappa[to]$, and updating global variables according to the update vector $\vec{u}$.

A counter system $Sys(\mathsf{TA})$ of a multi-round TA is defined analogously. A configuration captures the values of location counters and global variables *in each round*, and parameter values (that do not change over rounds). Then we define that a transition is *unlocked in a round $R$* by evaluating the guard $\phi$ and the counter of location *from* in the round $R$. The execution of the transition in $\sigma$ accordingly updates $\kappa[from, R]$, $\kappa[to, R]$ and global variables of that round, while the values of these variables in other rounds stay unchanged.

**Linear temporal logic notations.** Following a standard model checking approach, we use formulas in linear temporal logic (LTL) [55] to formalize the desired properties of distributed algorithms. The basic elements of these formulas, called atomic propositions, are predicates over configurations related (i) to the emptiness of each location at each round and (ii) to the evaluation of threshold guards in each round. They have the following form: (i) $\kappa[L, R] \neq 0$ expresses that at least one correct process is in location $L$ in round $R$, while $\kappa[L, R] = 0$ expresses the opposite (in one-round systems we just write $\kappa[L] \neq 0$ or $\kappa[L] = 0$); (ii) the evaluation of $[b_0, R]{\geq}2t{+}1{-}f$ depends on the values of the shared variable $b_0$ in round $R$ and parameters $t$ and $f$ (in one-round systems we just write $b_0{\geq}2t{+}1{-}f$). LTL builds on propositional logic with $\Rightarrow$ for 'implication', $\vee$ for 'or' and $\wedge$ for 'and', and has extra temporal operators $\Diamond$ standing for 'eventually' and $\Box$ for 'always'. LTL formulas are evaluated over infinite runs of $Sys(\mathsf{TA})$. Examples of LTL properties in a one-round system are (*BV-Just*$_v$), (*BV-Obl*$_v$) and (*BV-Unif*$_v$) (see page 8). LTL properties in multi-round systems often have quantifiers over round variables, as for example in (*Agree*$_v$) and (*Valid*$_v$) (see page 12).

The tool ByMC is used to automatically verify a specific fragment of LTL on one-round systems [35, 36], which is sufficient to express safety and liveness properties of consensus [10]. Moreover, thanks to communication-closure, the verification for this fragment of temporal logic on multi-round systems reduces to one-round systems [10, Theorem 6] (see also Appendix A).

The assumption of reliable communication is modeled as follows at the TA level: if the guard of a rule is true infinitely often, then the origin location of that rule will eventually be empty. This reflects that an if branch of the pseudo-code is taken if the condition is true. This *progress assumption* is in particular crucial to prove liveness properties: in the sequel, we prepend it to the liveness properties in the TA specification.

## 3 THE BINARY VALUE BROADCAST

To overcome the limited scalability of model checking tools, our holistic verification approach consists of decomposing a distributed algorithm into encapsulated components of pseudocode that can be modelled in threshold automata and verified in isolation to obtain a simplified threshold automaton that is amenable to automated verification.

In this section we focus on a *binary value broadcast*, or bv-broadcast for short, that will serve as the main building block of the Byzantine consensus algorithm of Section 4. In Section 3.1 we formally model the bv-broadcast algorithm pseudocode as a threshold automaton that tolerates a number $f$ of Byzantine failures upper-bounded by $t$ among $n$ processes. In Section 3.2 we model the specification of bv-broadcast in LTL and verify, within 10 seconds, that it holds. In Section 3.3 we introduce the fairness of an infinite sequence of executions of bv-broadcast that will play a crucial role in verifying holistically in Section 5 the Byzantine consensus algorithm.

### 3.1 Modeling the binary value broadcast pseudocode into a threshold automaton

The binary value broadcast [47], or *bv-broadcast* for short, is a communication primitive guaranteeing that all binary values “bv-delivered” were “bv-broadcast” by a correct process. It is particularly useful to solve the Byzantine consensus problem with randomization [15, 48] or partial synchrony [14, 19]. As discussed before, Figures 1 and 2 in Section 2 give its pseudocode and the corresponding threshold automaton, respectively. We now explain how we model our bv-broadcast pseudocode (Fig. 1) parameterized by $n$ and $f$ into a threshold automaton (Fig. 2) using the synthesis methodology [41].

**Pseudocode of the binary value broadcast.** The bv-broadcast algorithm pseudocode (Fig. 1) aims at having at least $2t$+1 processes broadcasting the same binary value. Each process starts this algorithm in one of two states, depending on its input value 0 or 1. Once a correct process receives a value from $t$+1 distinct processes, it broadcasts it (line 4) if it did not do it already (line 4); broadcast is not Byzantine fault tolerant and just sends a message to all the other processes. Once a correct process receives a value from $2t$+1 distinct processes, it delivers it. Here the delivery at process $p_i$ is modeled by adding the value to the set *contestants*, which will simplify the pseudocode of the Byzantine consensus algorithm in Section 4.1.

**Threshold automaton of the binary value broadcast.** To match the two initial states from which a process starts the bv-broadcast algorithm, we start the corresponding TA of Fig. 2 with two initial locations $V_0$ or $V_1$, indicating whether the (correct) process initially has value 0 or 1, resp. We can see form the pseudocode (Fig. 1) that a correct process $p_i$ sends only two types of messages, $(\mathsf{BV}, \langle 0, i \rangle)$ and $(\mathsf{BV}, \langle 1, i \rangle)$, these trigger the corresponding receptions at other processes. We thus define in the TA (Fig. 2) two global variables $b_0$ and $b_1$, resp., to capture the number of the two types of messages sent by correct processes. Thus, for example, $b_{0^{++}}$ models a process broadcasting message $(\mathsf{BV}, \langle 0, i \rangle)$. Because the algorithm only counts messages regardless of sender identities, we replace the messages from the pseudocode into $b_0$ and $b_1$ shared variables that are increased whenever a message is sent.

**From local to global variables for model checking.** While producing a formal model, extra care is needed to avoid introducing redundancies. For example, line 4 indicates that the process broadcasts value $v$ if it received $v$ from $t$+1 distinct processes. Instead of maintaining local receive variables, it is sufficient to enable a guard based on global send variables. Indeed, to remove redundant local receive variables, one can use the quantifier elimination for Presburger arithmetic [56] and obtain quantifier-free guard expressions over the shared variables that are valid inputs to ByMC [34, 38]. For more details, note that Stoilkovska et al. [61] eliminated the quantifier over the similar receive variables in Ben-Or's consensus algorithm [8] with the SMT solver Z3 [21]. Finally, the point-to-point reliable channels ensure that $p_j$ sends message $m$ to $p_i$ implies that eventually $p_i$ receives message $m$ from $p_j$. Hence shared variables $b_0$ and $b_1$ of the TA denote, respectively, the number of messages $(\mathsf{BV}, \langle 0, i \rangle)$ and $(\mathsf{BV}, \langle 1, i \rangle)$ *sent* by correct processes in the pseudocode.

**Modeling arbitrary (Byzantine) behaviors in the TA.** In order to model that, among the received messages, $f$ messages could have been sent by Byzantine processes, we need to map the 'if' statement of the pseudocode, comparing the number of receptions from distinct processes to $t$+1, to the TA guards, comparing the number $b_1$+$f$ of messages sent to $t + 1$. As $b_1$ counts the messages *sent* by correct processes and $f$ is the number of faulty processes that can send arbitrary values, a correct process can move from $B_0$ to $B_{01}$ as soon as $t$+1−$f$ correct processes have sent 1, provided that $f$ faulty processes have also sent 1. As a result, the guard of rule $r_4$ only evaluates over global send variables as: if more than $t$+1 messages of type $b_1$ have been *sent* by correct processes (hence the guard $b_1 \geq t{+}1{-}f$), then the shared variable $b_1$ is incremented, mimicking the *broadcast* of a new message of type $b_1$. Rule $r_3$ corresponds to lines 6–7 and delivers value $v = 0$ by storing it into variable *contestants* upon reception of this value from $2t + 1$ distinct processes. Hence, reaching location $C_0$ in the TA indicates that the value 0 has been delivered. As a process might stay in this location forever, we add a self-loop with guard condition set to true.

| locations | $V_0$ | $V_1$ | $B_0$ | $B_1$ | $B_{01}$ | $C_0$ | $CB_0$ | $C_1$ | $CB_1$ | $C_{01}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| val. broadcast | / | / | 0 | 1 | 0,1 | 0 | 0,1 | 1 | 0,1 | 0,1 |
| val. delivered | / | / | / | / | / | 0 | 0 | 1 | 1 | 0,1 |

Table 1. The locations of correct processes

**Other locations and rules.** The locations of the automaton correspond to the exclusive situations for a correct process depicted in Table 1. After location $C_0$, a process is still able to broadcast 1 and eventually deliver 1 after that. After location $B_{01}$, a process is able to deliver 0 and then deliver 1, or deliver 1 first and then deliver 0, depending on the order in which the guards are satisfied. Apart from the self-loops, note that the automaton is a directed acyclic graph.

Also, on every path in the graph, a shared variable is incremented only once. This reflects that in the pseudocode, a value may only be broadcast if it has not been broadcast before.

### 3.2 Properties of the binary value broadcast

As was previously proved by hand [47, 48], the bv-broadcast primitive satisfies four properties: BV-Justification, BV-Obligation, BV-Uniformity and BV-Termination. Here, we formalize these properties in linear temporal logic (LTL) to formally and automatically prove they hold. As we will discuss in Section 6, we verify them for any parameters $n$ and $t < n/3$ in less then 10 seconds.

The BV-Justification property states: "If $p_i$ is correct and $v \in \mathit{contestants}_i$, then $v$ has been bv-broadcast by some correct process" where $v \in \{0, 1\}$. Alternatively, "if $v$ is not bv-broadcast by some correct process and $p_i$ is correct, then $v \notin \mathit{contestants}_i$". In the TA from Fig. 2, $v \in \mathit{contestants}_i$ corresponds to process $i$ being in one of the locations $C_v$, $CB_v$ or $C_{01}$. Thus, justification can be expressed in LTL as the conjunction *BV-Just₀* ∧ *BV-Just₁* where, *BV-Just*$_v$ is the following formula:

$$\kappa[V_v] = 0 \;\Rightarrow\; \Box\left(\kappa[C_v] = 0 \wedge \kappa[CB_v] = 0 \wedge \kappa[C_{01}] = 0\right) \;. \qquad (\mathit{BV\text{-}Just}_v)$$

BV-Obligation requires that if at least $(t{+}1)$ correct processes bv-broadcast the same value $v$, then $v$ is eventually added to the set $\mathit{contestants}_i$ of each correct process $p_i$. This can again be formalized as $\mathit{BV\text{-}Obl}_0 \wedge \mathit{BV\text{-}Obl}_1$ where $\mathit{BV\text{-}Obl}_v$ is the following formula:

$$\Box\left(b_v \geq t{+}1 \;\Rightarrow\; \Diamond\big(\bigwedge_{L \in \mathrm{Locs}_v} \kappa[L] = 0\big)\right) \;, \qquad (\mathit{BV\text{-}Obl}_v)$$

where $\mathrm{Locs}_v = \{V_0, V_1, B_0, B_1, B_{01}, C_{1-v}, CB_{1-v}\}$ are all the possible locations of a process $i$ if $v \notin \mathit{contestants}_i$.

BV-Uniformity requires that if a value $v$ is added to the set $\mathit{contestants}_i$ of a correct process $p_i$, then eventually $v \in \mathit{contestants}_j$ at every correct process $p_j$. We formalize this as $\mathit{BV\text{-}Unif}_0 \wedge \mathit{BV\text{-}Unif}_1$ where $\mathit{BV\text{-}Unif}_v$ is the following:

$$\Diamond\left(\kappa[C_v]{\neq}0 \;\vee\; \kappa[CB_v]{\neq}0 \;\vee\; \kappa[C_{01}]{\neq}0\right) \;\Rightarrow\; \Diamond \bigwedge_{L \in \mathrm{Locs}_v} \kappa[L]{=}0 \;, \qquad (\mathit{BV\text{-}Unif}_v)$$

where $\mathrm{Locs}_v$ is defined as in ($\mathit{BV\text{-}Obl}_v$).

Finally, the BV-Termination property claims that eventually the set $\mathit{contestants}_i$ of each correct process $p_i$ is non empty. This can be phrased as the following LTL formula *BV-Term*:

$$\Diamond\left(\kappa[V_0]{=}0 \;\wedge\; \kappa[V_1]{=}0 \;\wedge\; \kappa[B_0]{=}0 \;\wedge\; \kappa[B_1]{=}0 \;\wedge\; \kappa[B_{01}]{=}0\right) \;, \qquad (\mathit{BV\text{-}Term})$$

forcing each correct process to be in one of the "final" locations $C_0, C_1, C_{01}, CB_0, CB_1$.

### 3.3 A fairness assumption to solve consensus

The traditional approach to establishing guarantee properties in verification is to require that all fair computations, instead of all computations, satisfy the property [4]. We thus introduce the fairness assumption that will be crucial in the rest of this paper. In order to define it, we first define a *good execution* of the bv-broadcast with respect to binary value $v$ as an execution:

*Definition 3.1 ($v$-good bv-broadcast).* A bv-broadcast execution is *$v$-good* if all its correct processes bv-deliver $v$ first.

We express this property in LTL. A bv-broadcast execution is *$v$-good* if no process ever visits locations $C_{1-v}$ and $CB_{1-v}$:

$$\Box\;\left(\kappa[C_{1-v}] = 0 \;\wedge\; \kappa[CB_{1-v}] = 0\right) \;.$$

Second, we consider an infinite sequence of bv-broadcast executions, tagged with $r \in \mathbb{N}$. It is important to stress that the setting is asynchronous, that is, processes invoke bv-broadcast infinitely many times, but at their own relative speed. Thus, they do not all invoke the bv-broadcast tagged with the same number $r$ at the same time. Nonetheless, every process invokes bv-broadcast infinitely many times and in the $r^{th}$ invocation its behavior depends on the messages sent in the $r^{th}$ invocation of other processes. Therefore, we refer to the $r^{th}$ execution of bv-broadcast even though the processes invoke it at different times.

*Definition 3.2 (fairness).* An infinite sequence of bv-broadcast executions is *fair* if there exists an $r$ such that the $r^{th}$ execution is $(r \bmod 2)$-good.

For simplicity, we use the terminology *fair* bv-broadcast when the infinite sequence of bv-broadcast executions is fair. We illustrate in Appendix B a possible execution of bv-broadcast whose existence implies fairness.

## 4 SIMPLIFIED AUTOMATON FOR BYZANTINE CONSENSUS

In this section we exploit the results of the first verification phase of Section 3 to simplify the threshold automaton of the Byzantine consensus algorithm. In Section 4.1 we introduce the pseudocode of the Byzantine consensus algorithm and its threshold automaton obtained with the naive modeling described in Section 3.1. In Section 4.2 we replace, in this threshold automaton, the inner bv-broadcast automaton by a smaller one obtained thanks to the bv-broadcast properties that are now verified. The verification of the resulting simplified automaton is deferred to Section 5.

### 4.1 The Byzantine consensus algorithm

Algorithm 1 is the DBFT Byzantine consensus algorithm [19] that relies on the fair binary value broadcast of Section 3. It is currently used in the Red Belly Blockchain, a recent blockchain that achieves unprecedented scalability [20]. More precisely, the DBFT binary consensus comes in two different variants: (i) a first variant that is safe but not live in the asynchronous setting, (ii) a second variant that is safe and live under the partial synchrony assumption. We use the first variant of it (without coordinator or timeout) here and show that it is live under our new fairness assumption. The DBFT binary consensus invokes bv-broadcast$(\cdot)$ at line 6 and uses a set *contestants* of binary values, whose scope is global, updated by the bv-broadcast (Fig. 1, line 7) and accessed by the procedure propose$(\cdot)$ (Alg. 1, line 7).

**Algorithm 1** The Byzantine consensus algorithm at process $p_i$

```
1:  Global scope variable:
2:    $contestants \subseteq \{0,1\}$, set of binary values, init. $\emptyset$.

3:  propose($est$):
4:    $r \leftarrow 0$
5:    repeat:
6:      bv-broadcast(EST, $\langle est, i \rangle$)
7:      wait until ($contestants \neq \emptyset$)
8:      broadcast(AUX, $\langle contestants, i \rangle$) $\rightarrow favorites$
9:      wait until $\exists c_1, \ldots, c_{n-t} : \forall 1 \leq j \leq n-t\ favorites[c_j] \neq \emptyset \wedge (qualifiers \leftarrow \cup_{\forall 1 \leq j \leq n-t} favorites[c_j]) \subseteq contestants$
10:     if $qualifiers = \{v\}$ then
11:       $est \leftarrow v$
12:       if $v = (r \bmod 2)$ then decide($v$)
13:     else $est \leftarrow (r \bmod 2)$
14:     $r \leftarrow r + 1$
```

As mentioned in Section 2, recall that the algorithm is communication-closed, so that for simplicity in the presentation we omit the current round number $r$ as the subscript of the variables and the parameter of the function calls. Variable *favorites* is an array of $n$ indices whose $j^{th}$ slot records, upon delivery, the message broadcast by process $j$ in the current round. Each process $p_i$ manages the following local variables: the current estimate *est*, initially the input value of $p_i$; and a set of binary values *qualifiers*. This algorithm maintains a round number $r$, initially 0 (line 4), and incremented at the end of each iteration of the loop at line 14. Process $p_i$ exchanges EST and AUX messages (lines 6–8), until it receives AUX messages from $n - t$ distinct processes whose values were bv-delivered by $p_i$ (line 9). Process $p_i$ then tries at line 12 to decide a value $v$ that depends on the content of *qualifiers* and the parity of the round. If *qualifiers* is a singleton there are two possible cases: if the value is the parity of the round then $p_i$ decides this value (line 12), otherwise it sets its estimate to this value (line 11). If *favorites* contains both binary values, then $p_i$ sets its estimate to the parity of the round (line 13). Although $p_i$ does not exit the infinite loop to help other processes decide, it can safely exit the loop after two rounds at the end of the second round that follows the first decision because all processes will be guaranteed to have decided. Note that even though a process may invoke decide$(\cdot)$ multiple times at line 12, only the first decision matters as the decided value does not change (see Section 5).

**The effect of fairness.** Note that the fairness notion from Section 3.3 ensures there is a round $r$ in which all correct processes bv-deliver $(r \bmod 2)$ first. The following lemma states that under the fairness assumption there is a round of Algorithm 1 in which all correct processes start with the same estimate. The proof is deferred to Appendix C.

LEMMA 4.1. *If the infinite sequence of bv-broadcast executions of Algorithm 1 is fair, with the $r^{th}$ execution being $(r \bmod 2)$-good, then all correct processes start round $r$+1 of Algorithm 1 with estimate $r \bmod 2$.*

**Modeling deterministic consensus.** Figure 3 depicts the threshold automaton (TA) obtained by modeling Algorithm 1 with the method we detailed in Section 3.1. The TA depicts two iterations of the repeat loop (line 5), since Algorithm 1 favors different values depending on the parity of the round number. For simplicity, we refer to the concatenation of

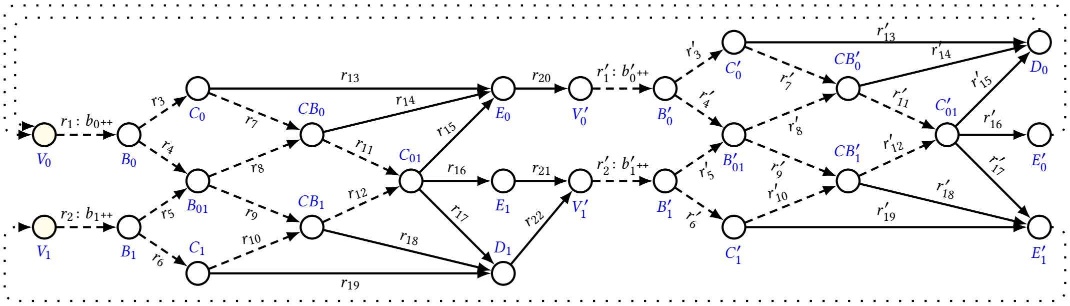

Fig. 3. The naive threshold automaton of the Byzantine consensus of Algorithm 1 where the embedded bv-broadcast automaton is depicted with dashed arrows. Precise formulations of all rules are in Appendix D. Note that the rules $r_{20}, r_{21}$ and $r_{22}$ represent transitions from the end of an odd round to the beginning of the following (even) round of Algorithm 1, while the dotted edges represent transitions from the end of an even round to the beginning of the following (odd) one.

two consecutive rounds of the algorithm as a *superround* of the TA. As one can expect, this TA embeds the TA of the bv-broadcast which is depicted by the dashed arrows, just as Algorithm 1 invokes the bv-broadcast algorithm of Fig. 1. We thus distinguish the outer TA modeling the consensus algorithm from the inner TA modeling the bv-broadcast algorithm. Although Algorithm 1 is relatively simple, the global TA happens to be too large to be verified through model checking, as we explain in Section 6; the main limiting factor is its 14 unique guards that constrain the variables to enable rules in the TA. The detail of each rule of the TA is deferred to Appendix D.

### 4.2 Simplified threshold automaton

Our objective is to formally prove that Algorithm 1 is unconditionally safe, and that it is live under the assumption of fairness at the bv-broadcast level. Since the threshold automaton of Figure 3 is too large to be handled automatically, we build on the properties proved for the bv-broadcast to simplify in the threshold automaton from Figure 3 the part representing the bv-broadcast. On the resulting simpler threshold automaton, assuming fairness of the bv-broadcast, we prove the termination of Algorithm 1 with the Byzantine model checker ByMC in Section 6.

**High-level idea.** Ideally, the simplified threshold automaton could be obtained from the one of Fig. 3 by merging all internal states of the bv-broadcast into a single state with two possible outcomes. However, such a merge is not trivial because the bv-broadcast procedure "leaks" into the consensus algorithm. First of all, line 7 of Algorithm 1 refers to contestants, a global variable that is modified by the bv-broadcast algorithm (Fig. 1). Second, a process can execute line 8 of Algorithm 1 even if the bv-broadcast has not terminated. To capture this porosity, we introduce a new shared variable, some additional states and a transition rule that exploits a correctness property of the bv-broadcast.

A superround $R$ of the simplified automaton from Fig. 4 captures round $2R{-}1$ followed by round $2R$ of Algorithm 1. One can thus restate Lemma 4.1 as the following corollary in the TA terminology. The proof is deferred to Appendix E.

Corollary 4.2. *Let $r \in \mathbb{N}$ be such that the $r^{th}$ execution of* bv-broadcast *in Algorithm 1 is* $(r \bmod 2)$*-good. Then:*

- *If there exists $R \in \mathbb{N}$ with $r = 2R{-}1$, then $\Box\left(\kappa[M_0, R] = 0\right)$ holds.*
- *If there exists $R \in \mathbb{N}$ with $r = 2R$, then $\Box\left(\kappa[M'_1, R] = 0\right)$ holds.*

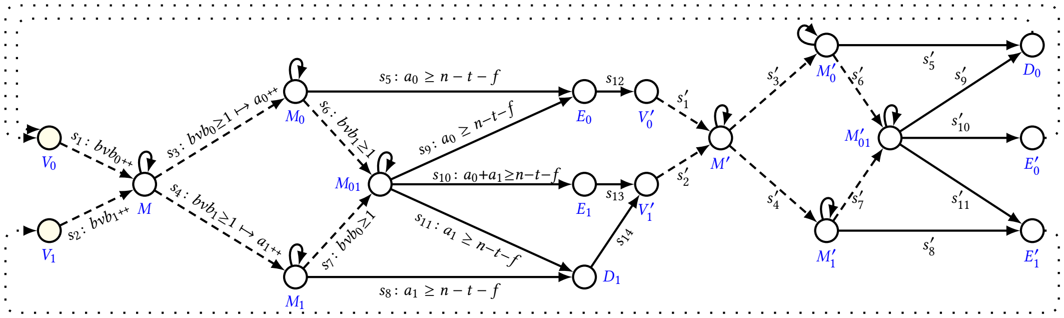

Fig. 4. The simplified threshold automaton of the Byzantine consensus of Algorithm 1 obtained after model checking the bv-broadcast. Rules $s'_j$, $1 \leq i \leq 11$, are obtained from $s_j$ by replacing each variable $c \in \{a_0, a_1, bvb_0, bvb_1\}$ with its corresponding one $c'$.

## 5 VERIFICATION OF BYZANTINE CONSENSUS

In this section we formally prove that Algorithm 1 solves the Byzantine consensus problem with the fair bv-broadcast and without partial synchrony. (Appendix B provides a counter-example illustrating why the algorithm does not terminate without the fair broadcast.) In particular, we apply a methodology developed for crash fault tolerant randomized consensus [10] to our context to prove both the safety (Section 5.1) and liveness (Section 5.2) properties of the deterministic Byzantine consensus algorithm.

### 5.1 Safety

Under no fairness assumption, one can prove the safety properties—agreement and validity—of the Byzantine consensus based on bv-broadcast. Precisely, we formulate these properties in LTL and want to establish that they hold on the threshold automaton of Fig. 4.

Agreement requires that no two correct processes disagree, that is, if one process decides $v$ then no process should decide $1{-}v$ for all binary values $v \in \{0, 1\}$. Thus, we want to prove that the following formula holds for both values of $v$:

$$\forall R \in \mathbb{N}, \forall R' \in \mathbb{N} \left( \Diamond\, \kappa[D_v, R] \neq 0 \implies \Box\, \kappa[D_{1-v}, R'] = 0 \right) , \qquad (Agree_v)$$

stating that for any two superrounds $R$ and $R'$, if eventually a process decides $v$, then globally (in any superround) no process will decide $1 - v$. In terms of the TA from Fig. 4, if a process enters location $D_v$ no process should enter location $D_{1-v}$ (not only in that superround, but in any other).

Validity requires that if no process proposes a value $v \in \{0, 1\}$, no process should ever decide that value. Hence, we want to prove the following formula for both values of $v$:

$$\forall R \in \mathbb{N} \left( \kappa[V_v, 1] = 0 \implies \Box\, \kappa[D_v, R] = 0 \right) , \qquad (Valid_v)$$

stating that if initially no process has value $v$, then globally (in any superround) no process decides $v$. In terms of the TA, if location $V_v$ is initially empty (in superround 1), then no process should enter location $D_v$ in any superround.

ByMC can only check formulas of the form $\forall R \in \mathbb{N}\ \varphi[R]$ (see Appendix A). Thus, automatically checking ($Agree_v$) and ($Valid_v$) is non-trivial, as they both involve two superround numbers: $R$ and $R'$ in ($Agree_v$), and 1 and $R$ in ($Valid_v$). We instead check well-chosen one-superround invariants ($Inv1_v$) and ($Inv2_v$):

$$\forall R \in \mathbb{N} \left( \Diamond\, \kappa[D_v, R] \neq 0 \implies \Box \left( \kappa[D_{1-v}, R] = 0 \wedge \kappa[E'_{1-v}, R] = 0 \right) \right) \ , \tag{$Inv1_v$}$$

$$\forall R \in \mathbb{N} \left( \Box\, \kappa[V_v, R] = 0 \implies \Box \left( \kappa[D_v, R] = 0 \wedge \kappa[E'_v, R] = 0 \right) \right) \ . \tag{$Inv2_v$}$$

The choice of these invariants follows a previous approach used for the crash fault tolerant consensus where it is shown that these invariants imply ($Agree_v$) and ($Valid_v$) [10, Proposition 2]. Intuitively, this follows from the fact that (i) emptiness of $D_0$ and $E'_0$ in one superround leads to the emptiness of $V_0$ in the next superround, and (ii) emptiness of $E'_1$ (and $D_1$) in one superround leads to the emptiness of $V_1$ in the next superround. Therefore, in order to prove agreement and validity, we only need to prove ($Inv1_v$) and ($Inv2_v$) for both values $v \in \{0, 1\}$. We successfully do this automatically with ByMC (see Section 6).

### 5.2 Liveness

We now aim at proving termination of Algorithm 1. First, we need to prove that every superround eventually terminates, in the sense that for every round eventually there are no processes in any location of that round to the exception of the final ones ($D_0$, $E'_0$ and $E'_1$). Formally, using ByMC we prove the following:

$$\forall R \in \mathbb{N}\, \Diamond \Big( \bigwedge_{L \in \mathcal{L} \setminus \{D_0, E'_0, E'_1\}} \kappa[L, R] = 0 \Big) \ . \tag{$SRoundTerm$}$$

From this property and the shape of the TA from Fig. 4, it easily follows that if no process ever enters $E'_0$ and $E'_1$ of some superround, then all processes visit $D_0$ in that superround. Similarly, if no process ever enters $E_0$ and $E_1$ of some superround, then all processes visit $D_1$ in that superround. This allows us to express termination as the following LTL property on the threshold automaton of Fig. 4:

$$\exists R \in \mathbb{N} \left( \Box \left( \kappa[E_0, R] = 0 \wedge \kappa[E_1, R] = 0 \right) \ \vee\ \Box \left( \kappa[E'_0, R] = 0 \wedge \kappa[E'_1, R] = 0 \right) \right) \ . \tag{$Term$}$$

In words, there is a superround $R$ in which either (i) all processes visit $D_1$, or (ii) all processes visit $D_0$. Here again formula (*Term*) is non-trivial to check since it contains an existential quantifier over superrounds, that cannot be handled by the model checker ByMC. Adapting the technique from [10, Section 7] to a non-randomized context, it is sufficient to prove a couple of properties on the threshold automaton of Fig. 4, that we detail below. The first property expresses that if no process starts a superround $R$ with value $v$, then all processes decide $1-v$ in superround $R$:

$$\begin{aligned} \forall R \in \mathbb{N} & \left( \Box \left( \kappa[V_0, R] = 0 \right) \Rightarrow \Box \left( \kappa[E_0, R] = 0 \wedge \kappa[E_1, R] = 0 \right) \right) \\ \wedge & \left( \Box \left( \kappa[V_1, R] = 0 \right) \Rightarrow \Box \left( \kappa[E'_0, R] = 0 \wedge \kappa[E'_1, R] = 0 \right) \right) \ . \end{aligned} \tag{$Dec$}$$

The second property claims that (i) emptiness of $M_0$ in superround $R$ implies (emptiness of $E_0$ and therefore also) emptiness of $D_0$ and $E'_0$ in $R$ and (ii) emptiness of $M'_1$ in superround $R$ implies emptiness of $E'_1$ in $R$:

$$\begin{aligned} \forall R \in \mathbb{N} \Big( & (\Box\, \kappa[M_0, R] = 0) \implies \Box\, (\kappa[D_0, R] \wedge \kappa[E'_0, R] = 0)) \\ & \wedge\ (\Box\, \kappa[M'_1, R] = 0) \implies \Box\, \kappa[E'_1, R] = 0) \Big) \ . \end{aligned} \tag{$Good$}$$

The main idea is to exploit the fairness of bv-broadcast, which ensures the existence of a round $r$ which is ($r$ mod 2)-good. Intuitively, the next superround $R = \lceil r/2 \rceil$ is the desired witness for (*Term*), namely the one in which all processes decide (not necessarily for the first time). We formalize this in our main result:

Theorem 5.1. *Assuming fairness of the* bv-broadcast*, Algorithm 1 terminates.*

Proof. First we prove formulas (*SRoundTerm*) and (*Dec*) and (*Good*) automatically using the model checker ByMC. Formula (*SRoundTerm*) guarantees that formula (*Term*) indeed expresses termination. Next, we show that formulas (*Dec*) and (*Good*) together imply (*Term*). Indeed, since we assume fairness of the bv-broadcast, from Corollary 4.2 we know that there is a superround $R$ in which one of the following two scenarios happen:

- $\Box\, \kappa[M_1', R] = 0$. In this case formula (*Good*) implies $\Box\, \kappa[E_1', R] = 0$. Note that the form of the (dotted) round-switch rules yield that no process starts the superround $R$+1 with value 1, that is, we have $\Box\, \kappa[V_1, R{+}1] = 0$. Then formula (*Dec*) implies $\Box\, \big(\kappa[E_0', R{+}1] = 0 \wedge \kappa[E_1', R{+}1] = 0\big)$, which makes formula (*Term*) true, that is, all processes visit $D_0$ in superround $R$+1.
- $\Box\, \kappa[M_0, R] = 0$. In this case formula (*Good*) implies $\Box\, \big(\kappa[D_0, R] \wedge \kappa[E_0', R] = 0)\big)$. Now the round-switch rules yield that no process starts the superround $R$+1 with value 0, that is, we have $\Box\, \kappa[V_0, R{+}1] = 0$. Then formula (*Dec*) implies $\Box\, \big(\kappa[E_0, R{+}1] = 0 \wedge \kappa[E_1, R{+}1] = 0\big)$, which satisfies formula (*Term*), that is, all processes visit $D_1$ in $R$+1.

As a consequence, our automated proofs of properties (*SRoundTerm*) and (*Dec*) and (*Good*) guarantee termination of Algorithm 1 under fairness of bv-broadcast. □

## 6 EXPERIMENTS

In this section, we model check the safety but also the liveness properties of Byzantine consensus for any parameters $t$ and $n > 3t$. In particular, we show that we formally verify the simplified representation of the blockchain consensus in less than 70 seconds, whereas we could not model check its naive representation.

**Experimental settings.** We used the parallelized version of ByMC 2.4.4 with MPI. The bv-broadcast and the simplified automaton were verified on a laptop with Intel® Core™ i7-1065G7 CPU @ 1.30GHz × 8 and 32 GB of memory. The naive Threshold Automaton (TA) timed-out even on a 4 AMD Opteron 6276 16-core CPU with 64 cores at 2300MHz with 64 GB of memory. *Good* and *Dec* are only relevant for the simplified automaton. The specification of the termination for ByMC is deferred to Appendix F.

**Results.** Table 2 depicts the time (6th column) it takes to verify each property (3rd column) automatically. In particular, it lists the TA (1st column) on which these properties were tested, as well as the size of these TA (2nd column) as the number of guards locations and rules they contain. A schema (4th column) is a sequence of unlocked guards (contexts) and rule sequences that is used to generate execution paths [36] whose average length appears in the 5th column. It demonstrates the efficiency of our approach as it allows to verify all properties of the Byzantine consensus automatically in less than 70 seconds whereas a non-compositional approach timed out. Although not indicated here, we also generated a counter-example of *Inv1*$_0$ for $n > 3t$ on the composite automaton in ~4 s.

## 7 RELATED WORK

Interactive theorem provers [57, 59, 66] already checked proofs of consensus algorithms used in the blockchain industry. In particular, Coq helped prove the Raft consensus algorithm [67], which is not Byzantine fault tolerant but part of crash fault tolerant distributed ledgers [5, 12], and the Byzantine consensus algorithm of the Algorand blockchain [3]. In addition, Dafny [30] proved MultiPaxos, a consensus algorithm that tolerates crash failures. Isabelle/HOL [52] was used to prove Byzantine fault tolerant algorithms [17] and was combined with Ivy to try to prove the Byzantine

| TA | Size | Property | # schemas | Avg. length | Time |
|---|---|---|---|---|---|
| bv-broadcast<br>(Fig. 2) | 4 unique<br>guards<br>10 locations<br>19 rules | *BV-Just*$_0$ | 90 | 54 | 5.61s |
| | | *BV-Obl*$_0$ | 90 | 79 | 6.87s |
| | | *BV-Unif*$_0$ | 760 | 97 | 27.64s |
| | | *BV-Term* | 90 | 79 | 6.75s |
| Naive<br>consensus<br>(Fig. 3) | 14 unique<br>guards<br>24 locations<br>45 rules | *Inv1*$_0$ | >100 000 | - | >24h |
| | | *Inv2*$_0$ | >100 000 | - | >24h |
| | | *SRound-Term* | >100 000 | - | >24h |
| Simplified<br>consensus<br>(Fig. 4) | 10 unique<br>guards<br>16 locations<br>37 rules | *Inv1*$_0$ | 6 | 102 | 4.68s |
| | | *Inv2*$_0$ | 2 | 73 | 4.56s |
| | | *SRound-Term* | 2 | 109 | 4.13s |
| | | *Good*$_0$ | 2 | 67 | 4.55s |
| | | *Dec*$_0$ | 2 | 73 | 4.62s |

Table 2. Although none of the properties of the naive blockchain consensus could be verified within a day of execution of the model checker, it takes about ∼4 s to verify each property on the simplified representation of the blockchain consensus. Overall it takes less than 70 seconds to verify both that the binary value broadcast and the simplified representation of the blockchain consensus are correct.

consensus protocol of the Stellar blockchain [44] as discussed in the introduction. Theorem provers check proofs, not the algorithms. Hence, one has to invest efforts into writing detailed mechanical proofs.

Specialized decision procedures are a way of proving consensus algorithms. They were used to prove Paxos [39], which could itself be used in the aforementioned crash fault tolerant distributed ledgers. Crash fault tolerant consensus algorithms were manually encoded with their invariants and properties to prove formulae using the Z3 SMT solver [24]. Decision procedures also proved the safety of Byzantine fault tolerant consensus algorithms when $f = t$ [9] but not their termination. Similarly, a proof by refinement of the safety of a Byzantine variant of Paxos was proposed [40] but its liveness is not proven. These decision procedures require the user to fit the specification into a suitable logical fragment.

Explicit-state model checking fully automates verification of distributed algorithms [31, 68]. It allows to check the reliable broadcast algorithm [32], a common component of various blockchain consensus algorithms [18, 19, 46]. TLC [68] checked a reduction of fault tolerant distributed algorithms in the Heard-Of model that exploits their communication-closed property [16]. And the agreement of consensus algorithms was proved in the asynchronous setting [53]. These explicit-state tools enumerate all reachable states and thus suffer from state explosion.

Symbolic model checkers [13] cope with this explosion by representing state transitions efficiently. NuSMV and SAT helped check consensus algorithms for up to 10 processes [64, 65]. Apalache [33] uses satisfiability modulo theories (SMT) to check inductive invariants and verify symbolic executions of TLA$^+$ specifications of the reliable broadcast and crash fault tolerant consensus algorithms but requires parameters to be fixed. These tools cannot be used to prove (or disprove) correctness for an arbitrary number of processes.

Parameterized model checking [23] works for an arbitrary number $n$ of processes [11]. Although the problem is undecidable [6] in general, one can verify specific classes of algorithms [27]. Indeed, distributed algorithms with a ring-based topology were checked with automata-theoretic method [2] and with Presburger arithmetics formulae verified by an SMT solver [58]. Bosco [60] has been the focus of various parameterized verification techniques [7, 41], however, it acts as a fast path wrapper around a separate correct consensus algorithm that remains itself to be proven. The condition-based consensus algorithm [49, 50] was verified [7] with the Byzantine model checker ByMC [34, 36, 38], only under the condition that the difference between the numbers of processes initialized with 0 and 1 differ by at least $t$. Recently, the crash fault tolerant Ben-Or consensus algorithm was proved correct with a probabilistic reasoning extension of ByMC [10]. In this paper, we also exploit ByMC but prove the Byzantine consensus algorithm [19] of an existing blockchain [20].

## 8 CONCLUSION

We presented the first formal verification of a blockchain consensus algorithm thanks to a new holistic approach. Previous attempts to formally verify the liveness of blockchain consensus consisted of verifying different parts of the consensus algorithm without verifying the sum of the parts. By modeling directly the pseudocode into a disambiguated threshold automaton we guarantee that the "actual" algorithm is verified. By model checking the threshold automaton without the need for user-defined invariants and proofs, we drastically reduce the risks of human errors. We believe that this holistic verification technique will help verify or identify bugs in other distributed algorithms based on various broadcast primitives.

## REFERENCES

[1] Ittai Abraham, Guy Golan Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. 2017. *Revisiting Fast Practical Byzantine Fault Tolerance.* Technical Report. arXiv. https://arxiv.org/abs/1712.01367

[2] C. Aiswarya, Benedikt Bollig, and Paul Gastin. 2018. An automata-theoretic approach to the verification of distributed algorithms. *Information and Computation* 259, 3 (2018), 305–327.

[3] Musab A. Alturki, Jing Chen, Victor Luchangco, Brandon M. Moore, Karl Palmskog, Lucas Peña, and Grigore Rosu. 2019. Towards a Verified Model of the Algorand Consensus Protocol in Coq. In *International Workshops on Formal Methods (FM'19).* 362–367.

[4] Rajeev Alur and Thomas A. Henzinger. 1994. Finitary fairness. In *Annual IEEE Symposium on Logic in Computer Science (LICS'94).* IEEE Computer Society Press, 52–61.

[5] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *ACM European Conference on Computer Systems (CCS'18).*

[6] Krzysztof R Apt and Dexter C Kozen. 1986. Limits for Automatic Verification of Finite-State Concurrent Systems. *Inform. Process. Lett.* 22, 6 (May 1986), 307–309.

[7] A. R. Balasubramanian, Javier Esparza, and Marijana Lazic. 2020. Complexity of Verification and Synthesis of Threshold Automata. In *International Symposium on Automated Technology for Verification and Analysis (ATVA'20).* 144–160.

[8] Michael Ben-Or. 1983. Another Advantage of Free Choice (Extended Abstract): Completely Asynchronous Agreement Protocols. In *Annual ACM Symposium on Principles of Distributed Computing (PODC'83).* 27–30.

[9] Idan Berkovits, Marijana Lazic, Giuliano Losa, Oded Padon, and Sharon Shoham. 2019. Verification of Threshold-Based Distributed Algorithms by Decomposition to Decidable Logics. In *International Conference on Computer Aided Verification (CAV'19).* 245–266.

[10] Nathalie Bertrand, Igor Konnov, Marijana Lazic, and Josef Widder. 2019. Verification of Randomized Consensus Algorithms Under Round-Rigid Adversaries. In *International Conference on Concurrency Theory (CONCUR'19).* 33:1–33:15.

[11] Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. 2015. *Decidability of Parameterized Verification.* Morgan & Claypool Publishers.

[12] Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. 2016. Corda: An Introduction. *R3 CEV, August* (2016).

[13] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. 1990. Symbolic Model Checking: 10ˆ20 States and Beyond. In *Annual Symposium on Logic in Computer Science (LICS'90).* 428–439.

[14] Christian Cachin, Daniel Collins, Tyler Crain, and Vincent Gramoli. 2020. Anonymity Preserving Byzantine Vector Consensus. In *European Symposium on Research in Computer Security (ESORICS'20).* 133–152.

[15] Christian Cachin and Luca Zanolini. 2020. *Asymmetric Byzantine Consensus.* Technical Report 2005.08795. arXiv.

[16] Mouna Chaouch-Saad, Bernadette Charron-Bost, and Stephan Merz. 2009. A Reduction Theorem for the Verification of Round-Based Distributed Algorithms. In *International Workshop on Reachability Problems (RP'09)*. 93–106.

[17] Bernadette Charron-Bost, Henri Debrat, and Stephan Merz. 2011. Formal Verification of Consensus Algorithms Tolerating Malicious Faults. In *SSS*. 120–134.

[18] Pierre Civit, Seth Gilbert, and Vincent Gramoli. 2021. Polygraph: Accountable Byzantine Agreement. In *Proceedings of the 41st IEEE International Conference on Distributed Computing Systems (ICDCS'21)*.

[19] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. 2018. DBFT: Efficient Leaderless Byzantine Consensus and its Applications to Blockchains. In *International Symposium on Network Computing and Applications (NCA'18)*. http://gramoli.redbellyblockchain.io/web/doc/pubs/DBFT-preprint.pdf

[20] Tyler Crain, Christopher Natoli, and Vincent Gramoli. 2021. Red Belly: A Secure, Fair and Scalable Open Blockchain. In *IEEE Symposium on Security and Privacy (S&P'21)*.

[21] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'08)*. 337–340.

[22] Stellar developers. [n.d.]. Caveat 3. https://github.com/stellar/scp-proofs Accessed: 13-02-2022.

[23] Rodney G. Downey and Michael R. Fellows. 1999. *Parameterized Complexity*. Springer.

[24] Cezara Dragoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. 2014. A Logic-Based Framework for Verifying Consensus Algorithms. In *International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'14)*. 161–181.

[25] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *J. ACM* 35, 2 (April 1988), 288–323.

[26] Tzilla Elrad and Nissim Francez. 1982. Decomposition of Distributed Programs into Communication-Closed Layers. *Science of Computer Programming* 2, 3 (1982), 155–173.

[27] E. Allen Emerson and Vineet Kahlon. 2000. Reducing Model Checking of the Many to the Few. In *International Conference on Automated Deduction (CADE'17)*. 236–254.

[28] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (April 1985), 374–382.

[29] Dana Fisman, Orna Kupferman, and Yoad Lustig. 2008. On Verifying Fault Tolerance of Distributed Protocols. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'08)*. 315–331.

[30] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Symposium on Operating Systems Principles (SOSP'15)*. 1–17.

[31] Gerard Holzmann. 2003. *The SPIN Model Checker*. Addison-Wesley.

[32] Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. 2013. Towards Modeling and Model Checking Fault-Tolerant Distributed Algorithms. In *International Symposium on Model Checking Software (SPIN'13) (LNCS)*, Vol. 7976. 209–226.

[33] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. 2019. TLA+ model checking made symbolic. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 123:1–123:30.

[34] Igor Konnov, Marijana Lazic, Ilina Stoilkovska, and Josef Widder. 2020. Tutorial: Parameterized Verification with Byzantine Model Checker. In *International Conference on Formal Techniques for (Networked and) Distributed Systems (FORTE'20)*. 189–207.

[35] Igor Konnov, Marijana Lazic, Helmut Veith, and Josef Widder. 2017. Para$^2$: parameterized path reduction, acceleration, and SMT for reachability in threshold-guarded distributed algorithms. *Formal Methods in System Design* 51, 2 (2017), 270–307.

[36] Igor Konnov, Marijana Lazic, Helmut Veith, and Josef Widder. 2017. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In *Symposium on Principles of Programming Languages (POPL'17)*. ACM, 719–734.

[37] Igor Konnov, Helmut Veith, and Josef Widder. 2017. On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. *Information and Computation* 252 (2017), 95–109.

[38] Igor Konnov and Josef Widder. 2018. ByMC: Byzantine Model Checker. In *ISoLA*. 327–342.

[39] Bernhard Kragl, Constantin Enea, Thomas A. Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. 2020. Inductive sequentialization of asynchronous programs. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'20)*. 227–242.

[40] Leslie Lamport. 2011. Byzantizing Paxos by Refinement. In *International Symposium on Distributed Computing (DISC'11)*. 211–224.

[41] Marijana Lazic, Igor Konnov, Josef Widder, and Roderick Bloem. 2017. Synthesis of Distributed Algorithms with Parameterized Threshold Guards. In *International Conference on Principles of Distributed Systems (OPODIS'17)*. 32:1–32:20.

[42] Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (1975), 717–721.

[43] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. 2019. Fast and Secure Global Payments with Stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. 80–96.

[44] Giuliano Losa and Mike Dodds. 2020. On the Formal Verification of the Stellar Consensus Protocol. In *Workshop on Formal Methods for Blockchains (FMBC@CAV'20)*. 9:1–9:9.

[45] Giuliano Losa, Eli Gafni, and David Mazières. 2019. Stellar Consensus by Instantiation. In *33rd International Symposium on Distributed Computing, DISC 2019 (LIPIcs)*, Jukka Suomela (Ed.), Vol. 146. 27:1–27:15.

[46] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *Conference on Computer and Communications Security (CCS'16)*.

[47] Achour Mostéfaoui, Hamouna Moumen, and Michel Raynal. 2014. Signature-free Asynchronous Byzantine Consensus with $T < N/3$ and $O(N^2)$ Messages. In *Symposium on Principles of Distributed Computing (PODC'14)*. 2–9.

[48] Achour Mostéfaoui, Hamouna Moumen, and Michel Raynal. 2015. Signature-free asynchronous binary Byzantine consensus with $t < n/3, O(n^2)$ messages and $O(1)$ expected time. *Journal of the ACM* (2015).

[49] Achour Mostéfaoui, Eric Mourgaya, Philippe Raipin Parvédy, and Michel Raynal. 2003. Evaluating the Condition-Based Approach to Solve Consensus. In *Dependable Systems and Networks (DSN'03)*. 541–550.

[50] Achour Mostéfaoui, Sergio Rajsbaum, and Michel Raynal. 2003. Conditions on Input Vectors for Consensus Solvability in Asynchronous Distributed Systems. *Journal of the ACM* 50, 6 (Nov. 2003), 922–954.

[51] Moni Naor and Udi Wieder. 2003. Scalable and Dynamic Quorum Systems. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*. 114–122.

[52] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, Vol. 2283. Springer.

[53] Tatsuya Noguchi, Tatsuhiro Tsuchiya, and Tohru Kikuno. 2012. Safety Verification of Asynchronous Consensus Algorithms with Model Checking. In *International Symposium on Dependable Computing (PRDC'12)*. 80–88.

[54] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. 1980. Reaching Agreement in the Presence of Faults. *Journal of the ACM* 27, 2 (1980), 228–234.

[55] Amir Pnueli. 1977. The Temporal Logic of Programs. In *IEEE Annual Symposium on Foundations of Computer Science (FOCS'77)*. 46–57.

[56] Mojżesz Presburger. 1929. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du congrès de Mathématiciens des Pays Slaves*. 92–101.

[57] Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. 2015. Formal Specification, Verification, and Implementation of Fault-Tolerant Systems using EventML. *Electronic Communication of the European Association of Software Science and Technology* 72 (2015).

[58] Arnaud Sangnier, Nathalie Sznajder, Maria Potop-Butucaru, and Sébastien Tixeuil. 2020. Parameterized verification of algorithms for oblivious robots on a ring. *Formal Methods in System Design* 56, 1 (2020), 55–89.

[59] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and proving with distributed protocols. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 28:1–28:30.

[60] Yee Jiun Song and Robbert van Renesse. 2008. Bosco: One-Step Byzantine Asynchronous Consensus. In *International Symposium on Distributed Computing (DISC'08)*. 438–450.

[61] Ilina Stoilkovska, Igor Konnov, Josef Widder, and Florian Zuleger. 2020. Eliminating Message Counters in Threshold Automata. In *International Symposium on Automated Technology for Verification and Analysis (ATVA'20)*. 196–212.

[62] Pierre Sutra. 2020. On the correctness of Egalitarian Paxos. *Inform. Process. Lett.* 156 (2020), 105901. https://doi.org/10.1016/j.ipl.2019.105901

[63] Pierre Tholoniat and Vincent Gramoli. 2022. Formal Verification of Blockchain Byzantine Fault Tolerance. In *Handbook on Blockchain*, Duc A. Tran, My T. Thai, and Bhaskar Krishnamachari (Eds.). Springer Nature.

[64] Tatsuhiro Tsuchiya and André Schiper. 2008. Using Bounded Model Checking to Verify Consensus Algorithms. In *Distributed Computing*, Gadi Taubenfeld (Ed.). 466–480.

[65] Tatsuhiro Tsuchiya and André Schiper. 2011. Verification of consensus algorithms using satisfiability solving. *Distributed Computing* 23, 5-6 (2011), 341–358.

[66] Klaus von Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. 2019. Pretend synchrony: synchronous verification of asynchronous distributed programs. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 59:1–59:30.

[67] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'15)*. 357–368.

[68] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model Checking TLA$^+$ Specifications. In *CHARME*. 54–66.

## A REDUCING MULTI-ROUND TA TO ONE-ROUND TA

Let us first formally define a (finite or infinite) *run* in a (one-round or multi-round) counter system *Sys*(TA). It is an alternating sequence of configurations and transitions $\sigma_0, t_1, \sigma_1, t_2, \ldots$ such that $\sigma_0 \in I$ is an initial configuration and for every $i \geq 1$ we have that $t_i$ is unlocked in $\sigma_{i-1}$, and executing it leads to $\sigma_i$, denoted $t_i(\sigma_{i-1}) = \sigma_i$.

Here we briefly describe the reasoning behind the reduction of multi-round TAs to one-round TAs [10, Theorem 6]. Note that the behavior of a process in one round only depends on the variables (the number of messages) of that round. Namely, we check if a transition is unlocked in a round by evaluating a guard and a location counter in that round. This allows us to modify a run by swapping two transitions from different rounds, as they do not affect each other, and preserve $\mathsf{LTL}_{\text{-}\mathsf{X}}$ properties, which are properties expressed in LTL without the next operator $\mathbf{X}$. The type of swapping we are interested in is the one where a transition of round $R$ is followed by a transition of round $R' < R$. Starting from any (fully asynchronous) run, if we keep swapping all such pairs of transitions, we will obtain a run in which processes synchronize at the end of each round and which has the same $\mathsf{LTL}_{\text{-X}}$ properties as the initial one. This, so-called *round-rigid* structure, allows us to isolate a single round and analyze it. Still, different rounds might behave differently as they have different initial configurations. If we have a formula $\forall R \in \mathbb{N}.\ \varphi[R]$, where $\varphi[R]$ is in the above mentioned fragment of (multi-round) LTL, then Theorem 6 of [10] shows exactly that it is equivalent to check that (i) this formula holds (or $\varphi[R]$ holds on all rounds $R$) on a multi-round TA, and (ii) formula $\varphi[1]$ (or just $\varphi$) holds on the one-round TA′ (naturally obtained from the TA by removing dotted round-switch rules) with respect to all possible initial configurations of all rounds. Thus, we can verify properties of the form $\forall R \in \mathbb{N}.\ \varphi[R]$ on multi-round threshold automata, by using ByMC to check $\varphi$ on a one-round threshold automaton with an enlarged set of initial configurations.

## B EXAMPLES OF FAIRNESS AND OF NON-TERMINATION WITHOUT FAIRNESS

First, we explain that the fairness is satisfied as soon as one execution of bv-broadcast has correct processes delivering all values broadcast by correct processes first. Then, we explain that the Byzantine consensus algorithm cannot terminate without an additional assumption, like fairness.

*Relevance of the fairness assumption.* It is interesting to note that our fairness assumption is satisfied by the existence of an execution with a particular reception order of some messages of the two broadcasts within the bv-broadcast. Consider that $t = \lceil n/3 \rceil - 1$ and that at the beginning of a round $r$, the two following properties hold: (i) estimate $r$ mod 2 is more represented than estimate $(1 - r)$ mod 2 among correct processes and (ii) all correct processes deliver the values broadcast by correct processes before any value broadcast during the bv-broadcast by Byzantine processes are delivered. Indeed, the existence of such a round $r$ in any infinite sequence of executions of bv-broadcast implies that this sequence is fair (Def. 3.2): as $r$ mod 2 is the only value that can be broadcast by $t$+1 correct processes, this is the first value that is received from $t$+1 distinct processes and rebroadcast by the rest of the correct processes. This is thus also the first value that is bv-delivered by all correct processes.

*Non-termination without fairness.* It is interesting to note why Algorithm 1 does not solve consensus when $t < n/3$ and without our fairness assumption. We exhibit an example of execution of the algorithm with $n = 4$ and $f = 1$, starting at round $r$ and for which the estimates of the correct processes are kept as $(1 - r)$ mod 2, $(1 - r)$ mod 2, $r$ mod 2 in rounds $r$ and $r$+2. Repeating this while incrementing $r$ yields an infinite execution, so that the algorithm never terminates.

Lemma B.1. *Algorithm 1 does not terminate without fairness.*

Proof. Consider, for example, processes $p_1, p_2, p_3$ and $p_4$ where $p_4$ is Byzantine and where 0, 0, 1 are the input values of the correct processes $p_1, p_2, p_3$, respectively, at round 1. We show that at the beginning of round 2, $p_1, p_2, p_3$ have estimates 0, 1, 1. First, as a result of the broadcast (line 2), consider that $p_1$ and $p_2$ receive 0 from $p_1$, $p_2$ and $p_4$ so that $p_1, p_2$ bv-deliver 0. Second, $p_2$ and $p_3$ receive 1 from $p_3$, $p_4$ and finally $p_2$ so that $p_2, p_3$ bv-deliver 1. Third, $p_3$ receives 0 from $p_0$, $p_2$ and finally from $p_3$ itself, hence $p_3$ bv-delivers 0. Now we have: (a) $p_1, p_2, p_3$ bv-deliver 0, 0, 1 and (b) $p_2, p_3$ later bv-deliver 1 and 0, respectively. As a result of (a), we have $p_1, p_2$ broadcast, and say $p_4$ sends, $\langle \textsc{aux}, 0, \cdot \rangle$ so that $p_0$ receives these three messages, $p_1, p_2$ broadcast $\langle \textsc{aux}, 0, \cdot \rangle$, and say $p_4$ sends, $\langle \textsc{aux}, 1, \cdot \rangle$ to $p_2$ so that $p_2$ receives these messages, $p_1$ broadcasts $\langle \textsc{aux}, 0, \cdot \rangle$ while $p_3$ broadcasts, and say $p_4$ sends, $\langle \textsc{aux}, 1, \cdot \rangle$ so that $p_3$ receives these messages. Finally, by (b) we have $contestants_2 = contestants_3 = \{0, 1\}$. This implies that the $n - t$ first values inserted in

$\mathit{favorites}_1$, $\mathit{favorites}_2$ and $\mathit{favorites}_3$ in round $r$ are values $\{0\}$, $\{0,1\}$, $\{0,1\}$, respectively. Finally, $\mathit{qualifiers}_1$, $\mathit{qualifiers}_2$ and $\mathit{qualifiers}_3$ are $\{0\}$, $\{0,1\}$ and $\{0,1\}$, respectively. And $p_1, p_2, p_3$ set their estimate to $0, 1, 1$.

It is easy to see that a symmetric execution in round $r' = r + 1$ leads processes to change their estimate from $0, 1, 1$ to $0, 0, 1$ looping back to the state where $r \bmod 2 = 1$ and estimate are $(1 - r) \bmod 2, (1 - r) \bmod 2, r \bmod 2$. □

## C STARTING A ROUND WITH IDENTICAL ESTIMATE

Lemma C.1 (Lemma 4.1). *If the infinite sequence of bv-broadcast invocations of Algorithm 1 is fair, with the $r^{th}$ invocation (in round $r$) being $(r \bmod 2)$-good, then all correct processes start round $r$+1 of Algorithm 1 with estimate $r \bmod 2$.*

Proof. The argument is that all correct processes wait until a growing prefix of the bv-delivered values that are re-broadcast implies that there is a subset of favorites, called *qualifiers*, containing messages from $n - t$ distinct processes such that $\forall v \in \mathit{qualifiers}.\ v \in \mathit{contestants}$. As we assume that the infinite sequence of bv-broadcast invocations of Algorithm 1 is fair, with the $r^{th}$ invocation being $(r \bmod 2)$-good, then we know that in round $r$ for every pair of correct processes $p_i$ and $p_j$ we have $p_i.\mathit{qualifiers} \subseteq p_j.\mathit{qualifiers}$ or $p_j.\mathit{qualifiers} \subseteq p_i.\mathit{qualifiers}$. If $p_i.\mathit{qualifiers} = p_j.\mathit{qualifiers}$ for all pairs, then by examination of the code, we know that they will set their estimate *est* to the same value depending on the parity of the current round.

Consider instead, with no loss of generality, that $p_i.\mathit{qualifiers}$ is a strict subset of $p_j.\mathit{qualifiers}$ in round $r$. As their values can only be binaries, in $\{0, 1\}$, this means that $p_i.\mathit{qualifiers}$ is a singleton, say $\{w\}$. As all correct processes bv-deliver $r \bmod 2$ first, which is then broadcast into $p_i.\mathit{favorites}$, we have $w = r \bmod 2$ and $p_i$'s estimate becomes $r \bmod 2$ at line 11. As $p_j.\mathit{qualifiers}$ is $\{0, 1\}$, the estimate of $p_j$ is also set to $r \bmod 2$ but at line 13. □

## D LARGE TA

Table 3 details the rules for the first half of the threshold automaton from Fig. 3.

| Rules | Guard | Update |
|---|---|---|
| $r_1$ | *true* | $b_0$++ |
| $r_2$ | *true* | $b_1$++ |
| $r_3$ | $b_0 \geq 2t{+}1{-}f$ | $a_0$++ |
| $r_4$ | $b_1 \geq t{+}1{-}f$ | $b_1$++ |
| $r_5$ | $b_0 \geq t{+}1{-}f$ | $b_0$++ |
| $r_6$ | $b_1 \geq 2t{+}1{-}f$ | $a_1$++ |
| $r_{14}, r_{15}, r_{16}$ | $a_0 \geq n{-}t{-}f$ | — |
| $r_8$ | $b_1 \geq t{+}1{-}f$ | $b_1$++ |
| $r_9$ | $b_1 \geq 2t{+}1{-}f$ | $a_1$++ |
| $r_{10}$ | $b_0 \geq 2t{+}1{-}f$ | $a_0$++ |
| $r_{11}$ | $b_0 \geq t{+}1{-}f$ | $b_0$++ |
| $r_{12}$ | $b_1 \geq 2t{+}1{-}f$ | — |
| $r_{13}$ | $b_0 \geq 2t{+}1{-}f$ | — |
| $r_7, r_{18}, r_{19}$ | $a_1 \geq n{-}t{-}f$ | — |
| $r_{16}$ | $a_0 \geq n{-}t{-}f$ | — |
| $r_{17}$ | $a_0{+}a_1 \geq n{-}t{-}f$ | — |
| $r_{20}, r_{21}, r_{22}$ | *true* | — |

Table 3. The rules of the threshold automaton from Fig. 3. We omit self loops that have trivial guard *true* and no update.

## E MISSING PROOF OF COROLLARY 4.2

We restate here Corollary 4.2 and give its proof.

Corollary E.1. *Let $r \in \mathbb{N}$ be such that the $r^{th}$ execution of* bv-broadcast *in Algorithm 1 is* $(r \bmod 2)$*-good. Then:*

- *If there exists $R \in \mathbb{N}$ with $r = 2R-1$, then $\square\left(\kappa[M_0, R] = 0\right)$ holds.*
- *If there exists $R \in \mathbb{N}$ with $r = 2R$, then $\square\left(\kappa[M_1', R] = 0\right)$ holds.*

Proof. By definition of an $(r \bmod 2)$-good execution, we know that in this particular invocation of bv-broadcast, all correct processes bv-deliver $r \bmod 2$ first. It follows from Lemma 4.1, that all correct processes start the next round with estimate set to $r \bmod 2$. There are two cases to consider depending on the parity of the round: If $r \bmod 2 = 1$, then this is the first round of superround $R$, i.e., $r = 2R - 1$. As a result, $\square\left(\kappa[M_0, R] = 0\right)$. If $r \bmod 2 = 0$, then this is the second round of superround $R$, i.e., $r = 2R$. As a result, $\square\left(\kappa[M_1', R] = 0\right)$. □

## F SPECIFICATION OF THE TERMINATION PROPERTY IN THE SIMPLIFIED THRESHOLD AUTOMATON FOR CONSENSUS ALGORITHM

The reliable communication assumption and the properties guaranteed by the bv-broadcast are expressed as preconditions for *s_round_termination*. The progress conditions work exactly the same as in [10]. However, since the shared counters representing the bv-broadcast execution do not represent regular messages, we cannot directly use the reliable communication assumption. Instead, we use the properties of the bv-broadcast that we proved in a separate automaton.

In practice, instead of using progress preconditions on the bv-broadcast counters in *s_round_termination*, such as:

```
(locM  == 0 || bvb1 < 1) && (locM  == 0 || bvb0 < 1) &&
(locM1 == 0 || bvb0 < 1) && (locM0 == 0 || bvb1 < 1)
```

we use the following:

```
/* BV-Termination */
(locM    == 0) &&
/* BV-Obligation */
(locM1 == 0 || bvb0 < T + 1) && (locM0 == 0 || bvb1 < T + 1) &&
/* BV-Uniformity */
(locM1 == 0 || aux0 == 0) && (locM0 == 0 || aux1 == 0) &&
```

One can note that we do not use BV-Justification as a precondition in this specification. Instead, the BV-Justification property is baked in the structure of the simplified threshold automaton (in the guard of the transition $M \longrightarrow M0, M1$).

The complete specification of the termination property follows:

```
s_round_termination:
<>[](
    (locV0   == 0) &&
    (locV1   == 0) &&

    /* BV-Termination */
    (locM    == 0) &&
    /* BV-Obligation */
    (locM1 == 0 || bvb0 < T + 1) &&
    (locM0 == 0 || bvb1 < T + 1) &&
    /* BV-Uniformity */
    (locM1 == 0 || aux0 == 0) &&
    (locM0 == 0 || aux1 == 0) &&

    /* Business as usual */
    (locM1   == 0 || aux1 < N - T) &&
    (locM0   == 0 || aux0 < N - T) &&
    (locM01  == 0 || aux0 + aux1 < N - T) &&

    (locD1   == 0) &&
    (locE0   == 0) &&
    (locE1   == 0) &&

    /* BV-Termination */
    (locMx    == 0 ) &&
    /* BV-Obligation */
    (locM1x   == 0 || bvb0x < T + 1) &&
    (locM0x   == 0 || bvb1x < T + 1) &&
    /* BV-Uniformity */
    (locM1x == 0 || aux0x == 0) &&
    (locM0x == 0 || aux1x == 0) &&

    (locM1x   == 0 || aux1x < N - T) &&
    (locM0x   == 0 || aux0x < N - T) &&
    (locM01x  == 0 || aux1x < N - T) &&
    (locM01x  == 0 || aux0x < N - T) &&
    (locM01x  == 0 || aux0x + aux1x < N - T)
    )
->
<>(
    locV0   == 0 &&
    locV1   == 0 &&
    locM    == 0 &&
    locM0   == 0 &&
    locM1   == 0 &&
    locM01  == 0 &&
    locE0   == 0 &&
    locE1   == 0 &&
    locD1   == 0 &&
    locMx   == 0 &&
    locM0x  == 0 &&
    locM1x  == 0 &&
    locM01x == 0
);

inv1_0: <>(locD0 != 0) -> [](locD1 == 0 && locE1x == 0);

inv2_0: [](locV0 == 0) -> [](locD0 == 0 && locE0x == 0);

inv1_1: <>(locD1 != 0) -> [](locD0 == 0 && locE0x == 0);

inv2_1: [](locV1 == 0) -> [](locD1 == 0 && locE1x == 0);

dec_0:  [](locV0 == 0) -> [](locE0 == 0 && locE1 == 0);

dec_1:  [](locV1 == 0) -> [](locE0x == 0 && locE1x == 0);

good_0: [](locM0 == 0) -> [](locD0 == 0 && locE0x == 0);

good_1: [](locM1x == 0) -> [](locE1x == 0);
```