Compositional Verification of Byzantine Consensus

Nathalie Bertrand, Vincent Gramoli, Igor Konnov, Marijana Lazic, Pierre Tholoniat, Josef Widder

Compositional Verification of Byzantine Consensus

Nathalie Bertrand
INRIA Rennes

Vincent Gramoli
University of Sydney and EPFL

Igor Konnov
Informal Systems

Marijana Lazić
TU Munich

Pierre Tholoniat
Columbia University

Josef Widder
Informal Systems

Abstract

Until now, computer-aided proofs of the liveness of byzantine consensus algorithms assumed synchrony to reason in lock steps or the error-prone manual intervention of experts in the proof checker but could not be automated through model checking.

We propose a compositional approach to verify a consensus algorithm, for any number n of processes and any upper bound $t < n/3$ on the number of byzantine processes. To this end, we identify a fairness property that makes this—otherwise purely asynchronous—byzantine consensus algorithm amenable to model checking. We decompose the algorithm in two parts: an inner broadcast algorithm and an outer decision algorithm. We encode these algorithms using threshold automata, and we formalize their properties in temporal logic. This allows us to automatically check the inner broadcasting algorithm, assuming fairness. For the verification of the outer algorithm, we simplify the automaton of the inner algorithm by relying on its checked properties. We verify in less than 70 seconds, not only the safety of byzantine consensus but also its liveness.

1 Introduction

Reasoning about executions of distributed algorithms is hard due to several sources of non-determinism, such as asynchrony and faults. It therefore requires expert knowledge to design and to rigorously prove distributed algorithms. Unsurprisingly, bugs in specifications and in proofs of theoretical work appear in the literature. With the resurgence of interest in byzantine consensus largely driven by blockchains, correctness of these algorithms has become crucial for security.

Thankfully, recent progress in automated verification are first steps towards the model checking of fault-tolerant consensus algorithms. For instance, parameterized model checking allows one to verify algorithms for an arbitrary number n of processes [12] that is unknown at design time. In addition, it reduces the model checking for any fault number f and its upper bound t to bounded model checking questions [33]. The threshold automaton (TA) framework for communication-closed algorithms [41, 7] exploits thresholds in guards such as "number of messages from distinct processes exceeds $2t + 1$", and in the resilience condition of the form $n > 3t$. The parameterized model checking of threshold automata builds upon a reduction [30, 47] that moves steps of asynchronous (interleaved) executions to obtain simpler executions, which are equivalent to the original executions with respect to safety and liveness properties. As moving steps preserves the admissibility of the execution, this reduction is, however, closely tied to the asynchronous semantics. Such a technique has recently proved instrumental in verifying fully asynchronous parts of consensus algorithms, like broadcast algorithms [41].

Due to the famous impossibility result [32], the above method could not be applied to proving deterministic consensus algorithms[1] in the asynchronous setting. The aforementioned reduction technique does not apply to partial synchrony [29] either: moving the message reception step to a later point in the execution might violate an assumed message delay. In fact, these delays are important as typical partially synchronous consensus algorithms feature monotonically increasing timers to catch up with the unknown bound on the delay to receive a message. The crux of the problem is to automatically prove liveness or the consensus termination. Due to similar complications, most known verification results are designed for either synchronous (lock step) or asynchronous semantics.

[1] We refer to a *deterministic consensus algorithm* as a consensus algorithm that satisfies linear temporal properties such as safety (agreement, validity) and liveness (termination), even if the environment (e.g., communication delays, scheduler) introduces non-determinism in the algorithm execution.

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 were devoted to proving the termination of partially synchronous consensus algorithms, like Paxos, assuming synchrony [35]. 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 [29].

The problem of verifying consensus is even more subtle when processes are byzantine as they can execute arbitrary steps, changing their state and the values they share. Hence, verifying that an algorithm tolerates byzantine faults requires to reason about the combination of asynchronous executions with all the possible scenarios resulting from arbitrary behaviors, hence adding up to the already large number of reachable states. The verification of such algorithms is thus often restricted to showing safety properties, like agreement and validity, and ignoring liveness [45]. This is not surprising, especially given that such coordinator-based protocols, need a non-trivial byzantine fault tolerant synchronizer algorithm [15].

1.1 Our results

In this paper, we leverage the automatic parameterized model checking to prove both the safety and liveness of a byzantine consensus algorithm. Our contributions are as follows:

1. We focus on a safe but not live variant of the binary byzantine consensus of DBFT [23] that is particularly simple: It does not need a coordinator, relies neither on randomization nor on signature and solves consensus deterministically. The novelty here lies in making it live by assuming a stronger notion of fairness (compared to typical reliable channels) to ensure termination, hence bypassing the need for partial synchrony. It builds upon ideas common to randomized and partially synchronous algorithms by featuring: (i) a binary value broadcast [50], a variant of the reliable broadcast for binary values that guarantees that correct processes deliver exclusively values broadcast by correct processes and (ii) a round-based execution that broadcasts and delivers values at multiple times before comparing the finally delivered message content to the parity of the round [23]. If the content matches the parity, then the algorithm decides, otherwise the algorithm updates its estimate for the next round. Interestingly, our fairness assumption only requires the existence of a specific ordering of message receptions in every infinite sequence of invocations of the binary value broadcast, without constraining the rest of the consensus algorithm.

2. We design a compositional proof methodology as a first step towards exploiting modularity of distributed algorithms in parameterized model checking. We provide threshold automata (TAs) models for two distributed algorithms that have strong interactions, namely an inner broadcast TA and an outer decision TA that invokes the inner one for some of its communications. To deal with the state space explosion, we express the guarantees of the inner broadcast primitive as temporal logic properties that we automatically verify with model checking and we replace the inner TA in the global TA by a gadget TA that captures the proven temporal specification. Hence, the compositional approach comprises a simple TA interface proved by hand and the combinatorial hard part that deals with asynchrony and faults proved automatically with the model checker.

3. We formally verify our consensus algorithm using the parameterized model checker (ByMC) [41] for any number n of processes and t of faulty processes. With ByMC, we check the temporal specifications (safety and liveness) of the inner TA encoding the broadcast algorithm, that we then exploit in the outermost TA. We demonstrate the efficiency gain of our compositional approach by running ByMC on (i) the naive TA encoding of our consensus algorithm as well as (ii) the composite TA resulting from our compositional approach. It turns out that ByMC could not prove the safety of the naive TA within days (after which we decided to forcefully stop it). In contrast, ByMC successfully checks the liveness and safety of the composite TA in slightly more than a minute.

Building upon recent progress in automated verification, our compositional proof is a typical example of new ways that can help addressing the error-prone task of proving distributed systems correct. While encoding partial synchrony in model checkers remains an open research challenge, our work shows that fairness, which appears as a more natural assumption that can be formalized for curent model checkers, can be strengthened to alleviate the need in distributed algorithms for additional assumptions, like partial synchrony.

1.2 Related Work

Interactive theorem provers [61, 59, 66] were used to prove consensus algorithms. In particular, Coq helped prove two-phase commit [61], Raft [67] and the Algorand consensus algorithm [4] while Dafny [35] proved MultiPaxos. Isabelle/HOL [54] was used to prove byzantine fault tolerant algorithms [21] and was combined with Ivy to prove the Stellar consensus protocol [48]. 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 [44]. Crash fault tolerant consensus algorithms were manually encoded with their invariants and properties to prove formulae using the Z3 SMT solver [27]. Decision procedures also proved the safety of byzantine fault tolerant consensus algorithms when $f = t$ [10] but not their termination. Similarly, a proof by refinement of the safety of a byzantine variant of Paxos was proposed [45] but its liveness is not proven. These decision procedures require the user to fit the specification into the suitable logical fragment.

Explicit-state model checking fully automates verification of distributed algorithms [36, 68]. It allows to check the reliable broadcast algorithm [37]. TLC [68] checked a reduction of fault tolerant distributed algorithms in the Heard-Of model that exploits their communication-closed property [20]. And the agreement of consensus algorithms was proved in the asynchronous setting [55]. These explicit-state tools enumerate all reachable states and thus suffer from state explosion.

Symbolic model checkers [16] cope with this explosion by representing state transitions efficiently. NuSMV and SAT helped check consensus algorithms for up to 10 processes [64, 65]. Apalache [38] 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 [26] works for an arbitrary number n of processes [12]. Although the problem is undecidable [6] in general, one can verify specific classes of algorithms [31]. Indeed, distributed algorithms with a ring-based topology were checked with automata-theoretic method [3] and with Presburger arithmetics formulae verified by an SMT solver [60]. Bosco [62] has been the focus of various parameterized verification techniques [46, 7], however, it acts as a fast path wrapper around a separate correct consensus algorithm. The condition-based consensus algorithm [53, 52] was verified [7] with the byzantine model checker ByMC [41, 43, 39], 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 [11]. In this paper, we also exploit ByMC but prove a byzantine consensus algorithm.

Some efforts were devoted to verify consensus algorithms in the partially synchronous setting [29] where after an unknown global stabilization time (GST) all links deliver messages in a bounded amount of time. Such algorithms were verified using parameterized model checking [49], however, these are only crash fault tolerant and cannot tolerate byzantine failures. PSync [28] views asynchronous executions in lock-steps and proves the LastVoting variant [22] of Paxos but requires semi-decision procedures of a fragment of first-order logic. Partial synchrony is often tied to some complexity in byzantine consensus algorithms. To terminate, partially synchronous algorithms typically distinguish the execution of a coordinator from the execution of other processes [19, 22, 45] and rely on a monotonically increasing timer to catch up with the unknown bound on the message delay. Our byzantine consensus algorithm does not inherit such intricacies.

Instead of assuming partial synchrony, we assume some notion of fairness. There exist related notions, like fair schedulers [14] and limited link synchrony [2]. Fair schedulers consider that the events of two processes receiving from two other processes are independent and that the probability for a process to receive from any other process is $\epsilon > 0$ in any round [14]. By contrast, our fairness is not probabilistic allowing us to model check safety and (deterministic) liveness. A key difference between our fairness assumption and partial synchrony is that our fairness does not impose restrictions on all links, which is similar to the notion of minimal synchrony needed to solve consensus [2]. This minimal synchrony was later named $\diamond[x+1]$-bi-source and helped solve byzantine consensus without all n^2 point-to-point links being eventually synchronous [1]. More precisely, $\diamond[x+1]$-bi-source states that there is a correct process that has $x+1$ bi-directional links with itself and other correct processes and these links eventually behave synchronously. It was shown in [8], that $(t+1)$-bi-source is necessary and sufficient to implement authenticated byzantine consensus. Later, the same result was generalized to unauthenticated byzantine consensus with $m \leq \lfloor (n-(t+1))/t \rfloor$ distinct values [13].

Finally, an interesting novelty of our algorithm is that it neither needs a coordinator (or leader) nor that any link be eventually synchronous. By contrast, all the consensus algorithms we know of that do not require all links to be timely rely on a coordinator [2, 1, 34, 13]. They use a rotating coordinator whose particular messages can influence

the estimate of other processes, to help them converge. If the coordinator does not manage to lead processes to a consensus, then another coordinator takes its role in what is called a new view. Before GST, processes may proceed at different rates. After GST, a synchronizer [15] is typically required to ensure that sufficiently many processes take part in the same view in order to guarantee termination. This is probably to circumvent this difficulty that the verification of the liveness of partially synchronous algorithms is often simplified by assuming synchrony [35].

The only work we know that assumes fairness for verification of asynchronous consensus is the one of finitary fairness [5]. Unfortunately, it is demonstrated in the shared memory context and without byzantine failures. One can see our contribution as a step forward in the message passing context and with byzantine failures, and in order to achieve this result we introduce a novel composition technique.

In Section 2 we introduce our preliminary definitions, in Section 3 we model our fair binary value broadcast, in Section 4 we present our composition, in Section 5 we verify the consensus algorithm and in Section 6 we present the results of the model checker and conclude. In the optional 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, F).

2 Preliminaries

The system is composed of n asynchronous sequential processes from the set $\Pi = \{p_1, \ldots, p_n\}$, and i is called the "index" of p_i. The processes communicate by exchanging messages through an asynchronous reliable point-to-point network, hence there is no bound on the delay to transfer a message but this delay is finite.

Failure model and fairness. Up to $t < n/3$ processes can exhibit a *byzantine* behavior [56], and behave arbitrarily. We refer to $f \leq t$ as the actual number of byzantine processes. A byzantine process is referred to as *faulty*, a non-faulty process is *correct*. As stated above, point-to-point reliable channels implicitly assume fairness, however, we will strengthen this fairness property by assuming that in an infinite sequence of binary value broadcast executions of our algorithm, there is one execution where some binary value is delivered by correct processes before the other value. The definition of this fairness is not needed for safety and is deferred to Section 3.3 to verify liveness.

Algorithm semantics. To define the asynchronous semantics of a distributed algorithm executed by these processes, we consider discrete time such that at each point in time, exactly one process takes a step. Hence the distributed execution is an interleaving of the individual steps taken by the processes. In particular, we assume that two messages cannot be received at the same time by the same process. Process p_i sends a message to p_j by invoking the primitive "send HEADER(m) to p_j", where HEADER indicates the type of message and m its content. Process p_i receives a message by executing the primitive "receive()". We refer to broadcast(HEADER$(m), \cdot, messages) \rightarrow messages$ as "for each $p_j \in \Pi$ do send HEADER(m) to p_j" and "upon reception of HEADER(m) from process p'_j do $messages[p'_j] \leftarrow messages[p'_j] \cup \{m\}$". We will use the process id i as a subscript to denote by var_i that a variable var is local to process i but we omit it when it is clear from the context.

The verification method considered in this paper exploits the fact that the algorithms are communication-closed [30], *i.e.* only messages from the current 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. We also assume that computation takes no time.

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* [42] describes the behaviour 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 $\langle \mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC \rangle$ where $\mathcal{L}$ is the set of locations, $\mathcal{I} \subset \mathcal{L}$ is the set of initial locations, Γ is the set of shared variables, Π 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 $\langle from, to, \phi, \vec{u} \rangle$, where *from* (resp. *to*) describes the source (resp. destination) locations, and the rule label is $\phi \mapsto \vec{u}$. Formula ϕ is called a *threshold guard* or simply a *guard*.

Example 1. *Fig. 1 in Section 3.1 presents the pseudocode of the binary value broadcast and its threshold automaton. There are 10 locations, namely* $\mathcal{L} = \{V_0, V_1, B_0, B_1, B_{01}, C_0, C_1, CB_0, CB_1, C_{01}\}$*, and two of them are initial,* $\mathcal{I} = \{V_0, V_1\}$*. Shared variables are* b_0 *and* b_1*, while parameter variables are* n*,* t *and* f*. The set of rules* $\mathcal{R}$ *consists of* $\{r_i \mid 1 \leq i \leq 12\}$ *and 7 self-loops. For instance, rule* r_3 *is defined as* $\langle B_0, C_0, b_0 \geq 2t+1-f, \vec{0} \rangle$*. Finally, the resilience condition is* $n > 3t \wedge t \geq f \geq 0$*.*

A *multi-round threshold automaton* is intuitively defined such that one round is represented by a threshold automaton, and additionally we have so-called *round-switch rules* that connect final locations with initial ones, and therefore allow processes to move from one round to the following one. We typically depict those round-switch rules as dotted arrows. Examples of multi-round TA are depicted in Figures 2 and 3. When it is clear from the context that automata have multiple rounds, we just call them threshold automata. When we want 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 Σ 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, therefore non-negative integers), values of global variables, and parameter values. A transition $t \in T$ is *unlocked in* σ if there exists a rule $r = \langle from, to, \phi, \vec{u} \rangle \in \mathcal{R}$ such that ϕ evaluates to true in σ, 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 σ 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 t is *unlocked in a round* R by evaluating the guard ϕ and the counter of location *from* in the round R. The execution of t in σ 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) [57] 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) $[b_0, R] \geq 2t+1-f$ is evaluated depending 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 $\Diamond$ that stands for 'eventually', $\Box$ for 'always'. LTL formulas are evaluated over infinite runs of $Sys(\mathsf{TA})$. Examples of LTL properties in a one-round system are ($BV\text{-}Just_v$), ($BV\text{-}Obl_v$) and ($BV\text{-}Unif_v$) (see page 7). LTL properties in multi-round systems often have quantifiers over round variables, as for example in ($Agree_v$) and ($Valid_v$) (see page 11).

The tool ByMC is used to automatically verify a specific fragment of LTL on one-round systems [40, 41]. This fragment is sufficient to express safety and liveness properties of consensus [11]. Moreover, using communication-closure, the verification for this fragment of temporal logic on multi-round systems reduces to one-round systems [11, Theorem 6]. We explain in more details this reduction in 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*[2] is in particular crucial to prove liveness properties: in the sequel, we prepend it to the liveness properties in the TA specification.

[2]The progress assumption is in the literature sometimes referred as fairness, but here we want to avoid confusion with the fairness from Section 3.

3 Fairness of binary value broadcast

To overcome the limited scalability of model checking tools, our compositional verification approach consists of decomposing a distributed algorithm in several blocks that can be verified in isolation to obtain a simplified threshold automaton that can be model checked.

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 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 to verify it within 10 seconds. 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 in Section 5 that we indeed solve the byzantine consensus problem.

3.1 Modeling the binary value broadcast

The binary value broadcast [50], 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 [51, 18] or partial synchrony [23, 17]. Consider Fig. 1 (left) that depicts its pseudocode and Fig. 1 (right) that depicts the corresponding threshold automaton (TA).

```
1: bv-broadcast(BV, val, i):
2:     broadcast(BV, ⟨val, i⟩)
3:     repeat:
4:         if (BV, ⟨v, ∗⟩) received from (t + 1) distinct processes but
5:                 not yet re-broadcast then
6:             broadcast(BV, v, i)
7:         if (BV, ⟨v, ∗⟩) received from (2t + 1) distinct processes
       then
8:             contestants ← contestants ∪ {v}
```

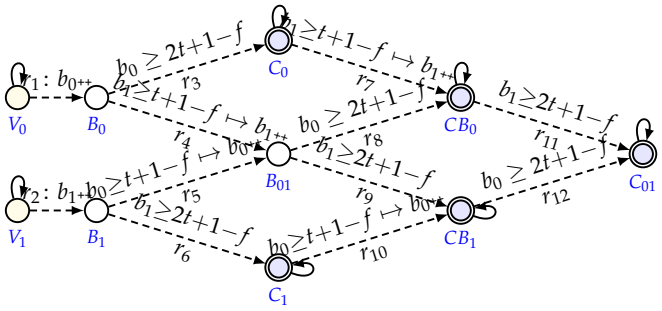

Figure 1: The pseudocode of the binary value broadcast (left) and its threshold automaton (right)

Pseudocode of the binary value broadcast

The bv-broadcast (see Fig. 1, left) aims at having at least $2t+1$ processes broadcasting the same binary value. Once a correct process receives a value from $t+1$ distinct processes, it broadcasts it (line 4) if it did not broadcast it already (line 5). 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 $contestants_i$, which will simplify the pseudocode of the byzantine consensus algorithm in Section 4.1.

Threshold automaton of the binary value broadcast

The corresponding TA of Fig. 1 (right) has two initial locations V_0 or V_1, indicating whether the (correct) process initially has value 0 or 1, respectively. We can see 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. Global variables b_0 and b_1, respectively, capture the number of the two types of messages sent by correct processes. Thus, for example, $b_{0{+}{+}}$ models a process broadcasting message $(\mathsf{BV}, \langle 0, i\rangle)$.

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. One may thus be tempted to evaluate a guard based on a local receive variable but, as the formal model needs to count sent values to not "re-broadcast" (line 5), it would be sufficient to simply enable a guard based on global send variables instead of also maintaining local receive variables. Note, however, that the point-to-point reliable channels ensures that p_j sends message m to p_i implies that eventually p_i receives message m from p_j. To remove redundant local receive variables, one can use the quantifier elimination for Presburger arithmetic [58] and obtain quantifier-free guard

expressions over the shared variables that are valid inputs to ByMC [43, 39]. For more details, note that Stoilkovska et al. [63] eliminated the quantifier over the similar receive variables in Ben-Or's consensus algorithm [9] with the SMT solver Z3 [25]. 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 7–8 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.

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.

locations	V_0	V_1	B_0	B_1	B_{01}	C_0	CB_0	C_1	CB_1	C_{01}
values broadcast	/	/	0	1	0,1	0	0,1	1	0,1	0,1
values delivered	/	/	/	/	/	0	0	1	1	0,1

Table 1: The locations of correct processes

3.2 Properties of the binary value broadcast

As was previously proved by hand [50, 51], 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 prove them correct. As we will discuss in Section 6, we verify the four properties automatically with model checking 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. 1, $v \in \mathit{contestants}_i$ 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 $\mathit{BV\text{-}Just}_0 \wedge \mathit{BV\text{-}Just}_1$ where, $\mathit{BV\text{-}Just}_v$ is the following formula:

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

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:

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

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:

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

where Locs_v is defined as in ($BV\text{-}Obl_v$).

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

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

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 asynchronous consensus

We now introduce a 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 where all correct processes (invoke bv-broadcast and) bv-deliver v before bv-delivering any other value. 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. We say that such an infinite sequence of bv-broadcast executions is *fair* if it contains an execution tagged with r that results in a good execution with respect to value $r \bmod 2$.

Definition 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}:

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

Definition 2 (fair infinite sequence of bv-broadcast executions)**.** *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.*

We simply refer to a *fair* bv-broadcast as if the infinite sequence of bv-broadcast executions is fair. For simplicity, we sometimes say that bv-broadcast is fair, when we actually mean that the infinite sequence of its executions is fair. We illustrate in Appendix B a possible execution of bv-broadcast whose existence implies fairness.

4 Composite 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 (non-compositional) modeling described in Section 3.1. In Section 4.2 we replace, in this threshold automaton, the inner bv-broadcast automaton by a smaller automaton simplified with the bv-broadcast properties that are now verified. The verification of the resulting composite automaton is deferred to Section 5.

4.1 The byzantine consensus algorithm

Algorithm 1 is a byzantine consensus algorithm that relies on the fair binary value broadcast of Section 3 and derives from a safe (but not live) variant of the binary consensus algorithm of DBFT [23] used in blockchains [24]. It 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(left), line 8) and accessed by the procedure propose$(\cdot)$ (Alg. 1, line 7).

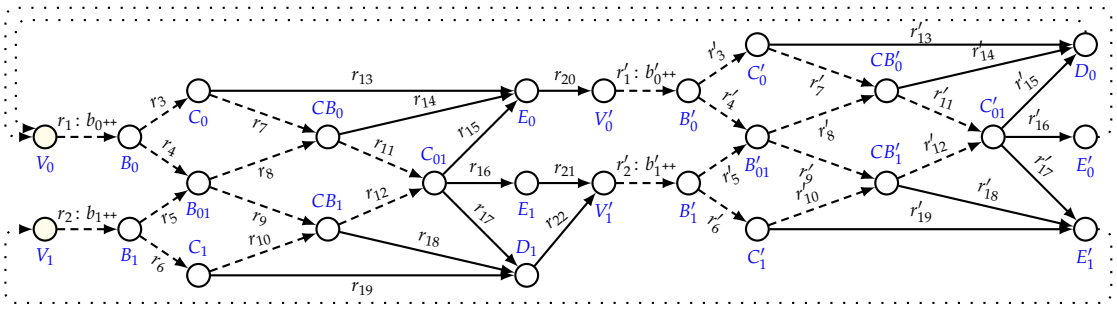

Figure 2: The naive threshold automaton of the byzantine consensus of Algorithm 1 where the embedded bv-broadcast automaton is depicted with dashed arrows. Precise formulations of all rules are in Appendix D. Note that the rules r_{20}, r_{21} and r_{22} represent transitions from the end of an odd round 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.

Algorithm 1 The byzantine consensus algorithm at p_i

```
1:  Global scope variable:
2:    contestants ⊆ {0,1} a set of binary values, initially ∅.

3:  propose(est):
4:    r ← 0
5:    repeat:
6:      bv-broadcast(EST, est, i)
7:      wait until (contestants ≠ ∅)
8:      broadcast(AUX, contestants, i) → favorites
9:      wait until ∃c_1, ..., c_{n−t} : ∀1 ≤ j ≤ n − t favorites[c_j] ≠ ∅ ∧
10:       (qualifiers ← ∪_{∀1≤j≤n−t} favorites[c_j]) ⊆ contestants
11:     if qualifiers = {v} then
12:       est ← v
13:       if v = (r mod 2) then decide(v)
14:     else est ← (r mod 2)
15:     r ← 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 15. Process p_i exchanges EST and AUX messages (lines 6–8), until it received AUX messages from $n - t$ distinct processes whose values were bv-delivered by p_i (lines 9–10). Process p_i then tries at line 13 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 13), otherwise it sets its estimate to this value (line 12). If *favorites* contains both binary values, then p_i sets its estimate to the parity of the round (line 14). 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 13, 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 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 2 depicts the threshold automaton (TA) obtained by modeling Algorithm 1 with the method of 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 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

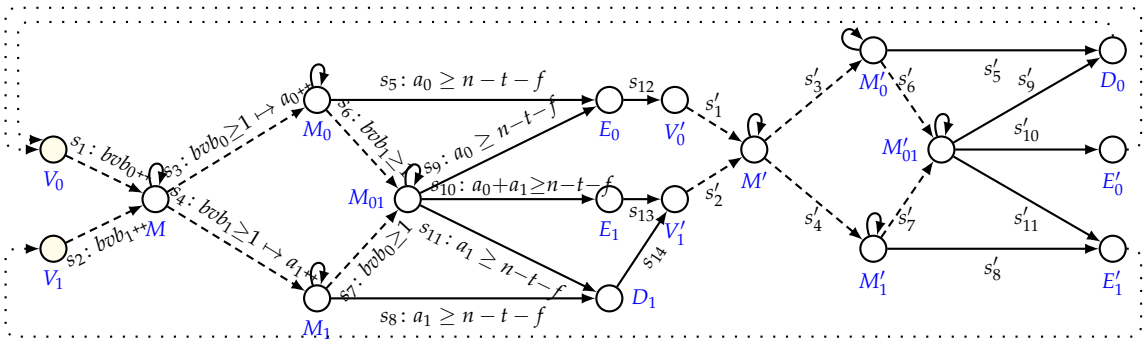

Figure 3: The composite threshold automaton of the byzantine consensus of Algorithm 1. Rules s'_j, $1 \leq i \leq 11$, are obtained from s_j by replacing each variable $c \in \{a_0, a_1, bvb_0, bvb_1\}$ with its corresponding one c'.

is depicted by the dashed arrows, just as Algorithm 1 invokes the bv-broadcast algorithm of Fig. 1 (left). 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 2 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 2 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. 2 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, left). 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.

Step by step construction. Figure 3 depicts the simplified composite threshold automaton of Algorithm 1 as a repeated superround. The left part of its superround corresponds to a round (or loop iteration) r_i of Algorithm 1 for a given correct process p_i such that $r_i \bmod 2 = 1$ whereas its right half corresponds to a round r_i where $r_i \bmod 2 = 0$. Below we describe the construction of the left half of Figure 3 by explaining its locations and rules:

- V_0, V_1. Initially, p_i holds an estimate binary value.
- $V_0, V_1 \to M$. Then, p_i invokes bv-broadcast with the value of its estimate (line 6). This transition has no guard. We count the number of correct processes that broadcast each value with shared variables bvb_0 and bvb_1.
- M. When p_i has called bv-broadcast but has not broadcast any AUX message yet, it is in location M. This corresponds to line 6. Note that a process that stays in M is not idle in practice: it actually keeps running the bv-broadcast primitive that was triggered earlier. However, we do not keep track of the messages exchanged in the underlying bv-broadcast primitive: bvb_0 and bvb_1 only count the initial value sent by bv-broadcast.
- $M \to M_0, M_1$. When $contestants_i$ becomes non-empty (line 7), p_i broadcasts an AUX message and proceeds to the next location. This AUX message contains the first value delivered to p_i by bv-broadcast. We can thus define two shared variables, a_0 and a_1, representing the number of 0s and 1s broadcast by correct processes with an AUX tag.

What is the guard on this condition? The earliest moment at which a correct process can have a non-empty *contestants* set is when another correct process has actually called bv-broadcast with such a value. Indeed, by *BV-Justification*, "If p_i is correct and $v \in contestants_i$, v has been bv-broadcast by a correct process". The condition that v has been bv-broadcast by a correct process is $bvb_v \geq 1$. Note that this condition relies only on the past behavior of correct processes, thanks to the byzantine fault tolerance of bv-broadcast. We do not have to take into account the possible messages sent by faulty processes here.

Such a transition may happen, but not necessarily immediately: even if another correct process has bv-broadcast 1 somewhere, p_i might not have delivered it yet and can stay longer in location M, thanks to the self-loop. The correctness properties will be checked on all these possible executions. To prove the termination of the consensus algorithm, we need the *BV-Termination* as a precondition.

- $M_0, M_1 \to M_{01}$. After having broadcast its AUX value, p_i might deliver the other value in $contestants_i$ thanks to the bv-broadcast primitive, and move to location M_{01}. The guard is the same as before. This transition does not trigger another AUX broadcast though. Recall that we assumed in Section 2 that no two receptions can occur at the same time at the same process.

- $M_0, M_1, M_{01} \to D_1, E_1, E_0$. Process p_i can pass the guard of line 10 when it has received enough AUX messages with the same value. Some messages might come from faulty processes because AUX messages are sent with a regular broadcast (not bv-broadcast). Hence, it might be possible for a correct process to move to the next step even if it has received only $n - t - f$ messages from correct processes.

A superround R of the composite automaton from Fig. 3 captures round $2R-1$ followed by round $2R$ of Algorithm 1. One can thus restate Lemma 1 as the following corollary in the TA terminology. The proof is deferred to Appendix E.

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

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 strategy used for crash fault tolerant randomized consensus [11] to our context to prove both the safety (Section 5.1) and liveness (Section 5.2) of the determinstic 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. 3.

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:

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

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. 3, 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:

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

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 ($Inv2_v$):

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

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

The choice of these invariants follows a previous approach used for the crash fault tolerant consensus [11] where Proposition 2 claims that correctness of these invariants implies correctness of ($Agree_v$) and ($Valid_v$). This easily 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 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. 3, 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. 3:

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

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 supperrounds, that cannot be handled by the model checker ByMC. Adapting the technique from [11, Section 7] to a non-randomized context, it is sufficient to prove a couple of properties on the threshold automaton of Fig. 3, 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:

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

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:

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

The main idea is to exploit the fairness of bv-broadcast, which ensures the existence of a round r which is $(r \bmod 2)$-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 1. *Assuming fairness of the* bv-broadcast, *the byzantine consensus algorithm (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 1 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 $\Box\left(\kappa[E_0', R{+}1] = 0 \wedge \kappa[E_1', R{+}1] = 0\right)$, 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 $\Box\left(\kappa[D_0, R] \wedge \kappa[E_0', R] = 0\right)$. 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 $\Box\left(\kappa[E_0, R{+}1] = 0 \wedge \kappa[E_1, R{+}1] = 0\right)$, 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. □

Threshold automaton	Size	Property	# schemas	Avg. schema length	Time
bv-broadcast (Fig. 1)	4 unique guards 10 locations 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 consensus (Fig. 2)	14 unique guards 24 locations 45 rules	*Inv1*$_0$	>100 000	-	>24h
		Inv2$_0$	>100 000	-	>24h
		SRound-Term	>100 000	-	>24h
Composite consensus (Fig. 3)	10 unique guards 16 locations 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: Experiments. We used the parallelized version of ByMC 2.4.4 with MPI. The bv-broadcast and the composite automaton were verified on a laptop with Intel® Core™ i7-1065G7 CPU @ 1.30GHz × 8 and 32 GB of memory. The naive automaton 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 composite automaton. Although not indicated here, we also generated a counter-example of *Inv1*$_0$ for $n > 3t$ on the composite automaton in ~4s. The specification of the termination for ByMC is deferred to Appendix F.

6 Experiments and Conclusions

Through composition, we model checked the safety but also the liveness of byzantine consensus for any parameters t and $n > 3t$. Table 2 demonstrates the relevance of our approach as it allows to verify the byzantine consensus automatically in less than 70 seconds whereas a non-compositional approach could not make it in less than a day.

Our results imply that our notion of fairness is a sufficient assumption to cope with the impossibility of solving consensus in asynchronous setting. In this sense, this work is a step towards addressing "the need for more refined models of distributed computing that better reflect realistic assumptions" that was raised as an open question in [32].

Acknowledgments

This research is supported under Australian Research Council Future Fellowship funding scheme (project number 180100496) entitled "The Red Belly Blockchain: A Scalable Blockchain for Internet of Things".

References

[1] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Consensus with byzantine failures and little system synchrony. In *DSN*, pages 147–155, 2006.

[2] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *PODC*, pages 328–337, 2004.

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

[4] Musab A. Alturki, Jing Chen, Victor Luchangco, Brandon M. Moore, Karl Palmskog, Lucas Peña, and Grigore Rosu. Towards a verified model of the algorand consensus protocol in coq. In *International Workshops on Formal Methods (FM)*, pages 362–367, 2019.

[5] Rajeev Alur and Thomas A. Henzinger. Finitary fairness. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 52–61. IEEE Computer Society Press, July 1994.

[6] Krzysztof R Apt and Dexter C Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, May 1986.

[7] A. R. Balasubramanian, Javier Esparza, and Marijana Lazic. Complexity of verification and synthesis of threshold automata. In *ATVA*, pages 144–160, 2020.

[8] Olivier Baldellon, Achour Mostéfaoui, and Michel Raynal. A necessary and sufficient synchrony condition for solving byzantine consensus in symmetric networks. In *ICDCN*, pages 215–226, 2011.

[9] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 27–30, 1983.

[10] Idan Berkovits, Marijana Lazic, Giuliano Losa, Oded Padon, and Sharon Shoham. Verification of threshold-based distributed algorithms by decomposition to decidable logics. In *CAV*, pages 245–266, 2019.

[11] Nathalie Bertrand, Igor Konnov, Marijana Lazic, and Josef Widder. Verification of randomized consensus algorithms under round-rigid adversaries. In *CONCUR*, pages 33:1–33:15, 2019.

[12] Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. *Decidability of Parameterized Verification*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015.

[13] Zohir Bouzid, Achour Mostéfaoui, and Michel Raynal. Minimal synchrony for byzantine consensus. In *PODC*, pages 461–470, 2015.

[14] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, October 1985.

[15] Manuel Bravo, Gregory V. Chockler, and Alexey Gotsman. Making byzantine consensus live. In *DISC*, pages 23:1–23:17, 2020.

[16] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10ˆ20 states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*, pages 428–439, 1990.

[17] Christian Cachin, Daniel Collins, Tyler Crain, and Vincent Gramoli. Anonymity preserving byzantine vector consensus. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS'20)*, pages 133–152, Sep 2020.

[18] Christian Cachin and Luca Zanolini. Asymmetric byzantine consensus. Technical Report 2005.08795, arXiv, 2020.

[19] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, November 2002.

[20] Mouna Chaouch-Saad, Bernadette Charron-Bost, and Stephan Merz. A reduction theorem for the verification of round-based distributed algorithms. In Olivier Bournez and Igor Potapov, editors, *Reachability Problems*, pages 93–106, 2009.

[21] Bernadette Charron-Bost, Henri Debrat, and Stephan Merz. Formal verification of consensus algorithms tolerating malicious faults. In *SSS*, pages 120–134, 2011.

[22] Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.

[23] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: Efficient leaderless Byzantine consensus and its applications to blockchains. In *NCA*, 2018.

[24] Tyler Crain, Christopher Natoli, and Vincent Gramoli. Evaluating the Red Belly Blockchain. Technical Report 1812.11747, arXiv, 2018.

[25] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[26] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999.

[27] Cezara Dragoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. A logic-based framework for verifying consensus algorithms. In *VMCAI*, pages 161–181, 2014.

[28] Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In *POPL*, pages 400–415, 2016.

[29] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[30] Tzilla Elrad and Nissim Francez. Decomposition of distributed programs into communication-closed layers. *Sci. Comput. Program.*, 2(3):155–173, 1982.

[31] E. Allen Emerson and Vineet Kahlon. Reducing model checking of the many to the few. In *Proceedings of the 17th International Conference on Automated Deduction (CADE-17)*, pages 236–254, 2000.

[32] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[33] Dana Fisman, Orna Kupferman, and Yoad Lustig. On verifying fault tolerance of distributed protocols. In *TACAS*, pages 315–331, 2008.

[34] Moumen Hamouma, Achour Mostéfaoui, and Gilles Trédan. Byzantine consensus with few synchronous links. In *OPODIS*, pages 76–89, 2007.

[35] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *SOSP*, pages 1–17, 2015.

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

[37] Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Towards modeling and model checking fault-tolerant distributed algorithms. In *SPIN*, volume 7976 of *LNCS*, pages 209–226, 2013.

[38] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. TLA+ model checking made symbolic. *Proc. ACM Program. Lang.*, 3(OOPSLA):123:1–123:30, 2019.

[39] Igor Konnov, Marijana Lazic, Ilina Stoilkovska, and Josef Widder. Tutorial: Parameterized verification with byzantine model checker. In *FORTE*, pages 189–207, 2020.

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

[41] Igor Konnov, Marijana Lazic, Helmut Veith, and Josef Widder. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In *POPL*, pages 719–734. ACM, 2017.

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

[43] Igor Konnov and Josef Widder. ByMC: Byzantine model checker. In *ISoLA*, pages 327–342, 2018.

[44] Bernhard Kragl, Constantin Enea, Thomas A. Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. Inductive sequentialization of asynchronous programs. In *PLDI*, pages 227–242, 2020.

[45] Leslie Lamport. Byzantizing paxos by refinement. In *DISC*, pages 211–224, 2011.

[46] Marijana Lazic, Igor Konnov, Josef Widder, and Roderick Bloem. Synthesis of distributed algorithms with parameterized threshold guards. In *OPODIS*, pages 32:1–32:20, 2017.

[47] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[48] Giuliano Losa and Mike Dodds. On the formal verification of the stellar consensus protocol. In *2nd Workshop on Formal Methods for Blockchains, FMBC@CAV 2020*, pages 9:1–9:9, 2020.

[49] Ognjen Maric, Christoph Sprenger, and David A. Basin. Cutoff bounds for consensus algorithms. In *CAV*, pages 217–237, 2017.

[50] Achour Mostéfaoui, Hamouna Moumen, and Michel Raynal. Signature-free asynchronous Byzantine consensus with $T < N/3$ and $O(N^2)$ messages. In *PODC*, pages 2–9, 2014.

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

[52] Achour Mostéfaoui, Eric Mourgaya, Philippe Raipin Parvédy, and Michel Raynal. Evaluating the condition-based approach to solve consensus. In *DSN*, pages 541–550, 2003.

[53] Achour Mostéfaoui, Sergio Rajsbaum, and Michel Raynal. Conditions on input vectors for consensus solvability in asynchronous distributed systems. *J. ACM*, 50(6):922–954, November 2003.

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

[55] Tatsuya Noguchi, Tatsuhiro Tsuchiya, and Tohru Kikuno. Safety verification of asynchronous consensus algorithms with model checking. In *IEEE 18th Pacific Rim International Symposium on Dependable Computing, PRDC 2012, Niigata, Japan, November 18-19, 2012*, pages 80–88, 2012.

[56] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[57] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[58] Mojżesz Presburger. Uber 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*, pages 92–101, 1929.

[59] Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. Formal specification, verification, and implementation of fault-tolerant systems using EventML. *ECEASST*, 72, 2015.

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

[61] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *PACMPL*, 2(POPL):28:1–28:30, 2018.

[62] Yee Jiun Song and Robbert van Renesse. Bosco: One-step byzantine asynchronous consensus. In *DISC*, pages 438–450, 2008.

[63] Ilina Stoilkovska, Igor Konnov, Josef Widder, and Florian Zuleger. Eliminating message counters in threshold automata. In *Proceedings of the 18th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 196–212, 2020.

[64] Tatsuhiro Tsuchiya and André Schiper. Using bounded model checking to verify consensus algorithms. In Gadi Taubenfeld, editor, *Distributed Computing*, pages 466–480, 2008.

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

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

[67] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*, pages 357–368, 2015.

[68] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA^+ specifications. In *CHARME*, pages 54–66, 1999.

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 $Sys(\mathsf{TA})$. 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 σ_{i-1}, and executing it leads to σ_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 [11, 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 $\mathsf{LTL}_{-\mathsf{X}}$ 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 $\mathsf{LTL}_{-\mathsf{X}}$ 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 [11] 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 φ) 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 φ 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 $r \bmod 2$ is more represented than estimate $(1 - r) \bmod 2$ 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. 2): as $r \bmod 2$ 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 $(1 - r) \bmod 2, (1 - r) \bmod 2, r \bmod 2$ in rounds r and $r+2$. Repeating this while incrementing r yields an infinite execution, so that the algorithm never terminates.

Lemma 2. *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, $\langle \text{AUX}, 0, \cdot \rangle$ so that p_0 receives these three messages, p_1, p_2 broadcast $\langle \text{AUX}, 0, \cdot \rangle$, and say p_4 sends, $\langle \text{AUX}, 1, \cdot \rangle$ to p_2 so that p_2 receives these messages, p_1 broadcasts $\langle \text{AUX}, 0, \cdot \rangle$ while p_3 broadcasts, and say p_4 sends, $\langle \text{AUX}, 1, \cdot \rangle$ 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 $favorites_1$, $favorites_2$ and $favorites_3$ in round r are values $\{0\}$,

$\{0,1\}$, $\{0,1\}$, respectively. Finally, $qualifiers_1$, $qualifiers_2$ and $qualifiers_3$ 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 3 (Lemma 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 $\forall v \in qualifiers.\ v \in contestants$. 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 $p_i.qualifiers \subseteq p_j.qualifiers$ or $p_j.qualifiers \subseteq p_i.qualifiers$. If $p_i.qualifiers = p_j.qualifiers$ 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 $p_i.qualifiers$ is a strict subset of $p_j.qualifiers$ in round r. As their values can only be binaries, in $\{0,1\}$, this means that $p_i.qualifiers$ is a singleton, say $\{w\}$. As all correct processes bv-deliver $r \bmod 2$ first, which is then broadcast into $p_i.favorites$, we have $w = r \bmod 2$ and p_i's estimate becomes $r \bmod 2$ at line 12. As $p_j.qualifiers$ is $\{0,1\}$, the estimate of p_j is also set to $r \bmod 2$ but at line 14. □

D Large TA

Table 3 details the rules for the first half of the threshold automaton from Fig. 2.

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. 2. We omit self loops that have trivial guard *true* and no update.

E Missing proof of Corollary 1

We restate here Corollary 1 and give its proof.

Corollary 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* $\square\,(\kappa[M_0, R] = 0)$ *holds.*
- *If there exists* $R \in \mathbb{N}$ *with* $r = 2R$, *then* $\square\,(\kappa[M_1', R] = 0)$ *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 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, $\square\,(\kappa[M_0, R] = 0)$. If $r \bmod 2 = 0$, then this is the second round of superround R, i.e., $r = 2R$. As a result, $\square\,(\kappa[M_1', R] = 0)$. □

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 [11]. 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 $M \to M0, M1$).

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);
```