Formal Verification of Blockchain Byzantine Fault Tolerance

Pierre Tholoniat and Vincent Gramoli

Abstract To implement a blockchain, the trend is now to integrate a non-trivial Byzantine fault-tolerant consensus algorithm instead of the seminal idea of waiting to receive blocks to decide upon the longest branch. After a dozen years of existence, blockchains trade now large amounts of valuable assets and a simple disagreement could lead to disastrous losses. Unfortunately, Byzantine consensus solutions used in blockchains are at best proved correct "by hand" as we are not aware of any of them having been automatically verified. We propose two contributions: (i) we illustrate the severity of the problem by listing six vulnerabilities of blockchain consensus including two new counter-examples; (ii) we then formally verify two Byzantine fault-tolerant components of Red Belly Blockchain (Crain et al. in Red belly: a secure, fair and scalable open blockchain, 2021, [32]) using the ByMC model checker. First, we specify its simple broadcast primitive in 116 lines of code that is verified in 40 s on a 2-core Intel machine. Then, we specify its blockchain consensus algorithm in 276 lines of code and assume a round-rigid adversary to verify in 17 minutes on a 64-core AMD machine using MPI. To conclude, we argue that it has now become both possible and crucial to formally verify the correctness of blockchain consensus protocols.

1 Introduction

As blockchain is a popular abstraction to handle valuable assets, it has become one of the cornerstones of promising solutions for building critical applications without requiring trust. Unfortunately, after a dozen years of research in the space, the blockchain still appears in its infancy, unable to offer the guarantees that are needed by the industry to automate critical applications in production. The crux of the problem

P. Tholoniat
Columbia University, New York, NY, USA
e-mail: pierre@cs.columbia.edu

V. Gramoli (✉)
University of Sydney, Sydney, Australia
e-mail: vincent.gramoli@sydney.edu.au

D. A. Tran et al. (eds.), *Handbook on Blockchain*, Springer Optimization and Its Applications 194, https://doi.org/10.1007/978-3-031-07535-3_12

is the difficulty of having remote computers agree on a unique block at a given index of the chain when some of them are malicious. The first blockchains [61] allow disagreements on the block at an index of the chain but try to recover from these disagreements before assets get stolen through double spending: with disagreement, an asset owner could be fooled when they observe that they received the asset. Instead the existence of a conflicting block within a different branch of the chain may indicate that the asset belongs to a different user who can re-spend it. This is probably why most blockchains now build upon some form of Byzantine fault-tolerant consensus solutions [17, 18, 31] that guarantee agreement despite malicious, also known as *Byzantine*, participants.

Solving the Byzantine consensus problem, defined four decades ago [65], is needed to guarantee that machines agree on a common block at each index of the chain. The consensus was recently shown to be necessary in the general scenario where conflicting transactions might be requested from distributed machines [41]. Various solutions to the consensus problem were proposed in the last four decades [8, 22, 30, 48, 49, 52, 69]. Most of these algorithms were proved correct "by hand", often listing a series of lemmas and theorems in prose leading the reader to the conclusion that the algorithm solves agreement, validity, and termination in all possible distributed executions. In the worst case, these algorithms are simply described with text on blog post [43, 52]. In the best case, a mathematical specification is offered, like in TLA+, but without machine-checked proofs [74]. Unfortunately, such a formal specification that is not machine-checked remains error prone [73].

Formal verification techniques are often limited while blockchain consensus protocols are complex and expected to run on hundreds or thousands of nodes. Theorem provers [3, 23, 53] check proofs but not algorithms. Proofs by refinement exist [50] but do not show liveness. Symbolic model checkers checked consensus algorithms but for up to 10 processes [75, 76]. Parameterized model checking [33] already proved Bosco [51], the Ben-Or consensus algorithm [12] and the condition-based consensus algorithm [9] for any number of processes. Unfortunately, Bosco [71] is a wrapper on top of another consensus that needs to be proven, Ben-Or's does not tolerate Byzantine failures and the condition-based consensus algorithm [59, 60] solves consensus only with specific sets of input values. As a result, none of these solutions fit blockchains. Only recently was a variant of the DBFT consensus algorithm proved live with any number of processes [11] using a decomposition.

In this paper, we first survey important problems that recently affected blockchain consensus. In particular, we propose two new counter-examples explaining why the Casper FFG algorithm, which should be integrated in phase 0 of Ethereum 2.0 and the HoneyBadgerBFT, which is being integrated into one of the most popular blockchain software, called `parity`, may not terminate. We also list four additional counter-examples from the literature to illustrate the amplitude of the problem for blockchains. While there exist alternative solutions to some of these problems that could be implemented it does not prevent other problems from existing. Moreover, proving "by hand" that the fixes solve the bugs may be found unconvincing, knowing that these bugs went unnoticed when the algorithms were proven correct, also "by hand", in the first place.

We then build upon modern tools and equipments at our disposal to formally verify components of the Red Belly Blockchain [32] consensus that do not assume synchrony under the assumption that $t < n/3$ processes are Byzantine (or *faulty*) among n processes. Red Belly Blockchain [32] is a fast blockchain that solves consensus deterministically and performs reasonably well on one thousand geodistributed replicas. Its scalability stems from the superblock optimization that combines multiple proposed blocks into one decision. Using Red Belly Blockchain as an example, we explain how the Byzantine model checker ByMC [47] can be used by distributed computing scientists to verify blockchain consensus components. The idea is to convert the distributed algorithm into a threshold automaton [51] that represents a state as a group of all the states in which a *correct* (or non-faulty) process resides until this process receives sufficiently many messages to transition. We offer the threshold automaton specification of a Byzantine fault-tolerant broadcast primitive that is key to few blockchains [28, 30, 56]. Finally, we also offer the threshold automaton specification of a slight variant of the Byzantine consensus algorithm [30] of Red Belly Blockchain that we prove safe and live under the round-rigidity assumption [13] that helps modeling a fair scheduler [15], hence allowing other distributed computing scientists to reproduce the verification with this publicly available model checker.

Various specification languages (e.g., [54, 79]) were proposed for distributed algorithms before threshold automata, but they did not allow the simplification needed to model check algorithms as complex as the Byzantine consensus algorithms needed in blockchain. As an example, in Input/Output Automata [54], the number of specified states accessible by an asynchronous algorithm before the threshold is reached could be proportional to the number of permutations of message receptions. Executing the automated verification of an invariant could require a computation proportional to the number of these permutations. More dramatically, the Byzantine fault model typically allows some processes to send arbitrarily formed and arbitrarily many messages—making the number of states to explore potentially infinite. As a result, this is only with the recent progress in parameterized model checking that we were able to verify our blockchain consensus components.

The remainder of the paper is organized as follows. Section 2 presents new and existing problems affecting known blockchain Byzantine consensus. In Sect. 3, we explain how we verified a Byzantine fault-tolerant broadcast abstraction common to multiple blockchains. In Sect. 4, we list the pseudocode, specification, and verification experiments of the Byzantine consensus used in Red Belly Blockchain. Section 5 presents the related work and Sect. 6 discusses our verifications and concludes the paper.

2 The Problem of Proving Blockchain Consensus Algorithms by Hand

In this section, we illustrate the risk of trying to prove blockchain consensus algorithms by hand by describing a list of safety and liveness limitations affecting the

Table 1 Some consensus algorithms that experienced liveness or safety limitations

Algorithms	Ref.	Limitation	Counter-example	Alternative	Blockchain
Randomized consensus	[57]	Liveness	[new]	[58]	HoneyBadger [56]
Casper	[18]	Liveness	[new]	[80]	Ethereum v2.0 [38]
Ripple consensus	[69]	Safety	[7]	[24]	xRapid [16]
Tendermint consensus	[17]	Safety	[6]	[5]	Tendermint [49]
Zyzzyva	[48]	Safety	[1]	[8]	SBFT [39]
IBFT	[52]	Liveness	[68]	[68]	Quorum [25]

Byzantine fault-tolerant algorithms implemented in actual blockchain systems. These limitations, depicted in Table 1, are not necessarily errors in the proofs but stem from the ambiguous descriptions in prose rather than formal statements and the lack of machine-checked proofs. As far as we know, until now no Byzantine fault-tolerant consensus algorithms used in a blockchain had been formally verified automatically.

2.1 The HoneyBadger and Its Randomized Binary Consensus

HoneyBadger [56] builds upon the combination of three algorithms from the literature to solve the Byzantine consensus with high probability in an asynchronous model. This protocol is being integrated in one of the most popular blockchain software, called Ethereum `parity`.[1] First, it uses a classic reduction from the problem of multi-value Byzantine consensus to the problem of binary Byzantine consensus working in the asynchronous model. Second, it reuses a randomized Byzantine binary consensus algorithm [57] that aims at terminating in expected constant time by using a common coin that returns the same unpredictable value at every process. Third, it uses a common coin implemented with a threshold signature scheme [19] that requires the participation of correct processes to return a value.

Randomized binary consensus. In each asynchronous round of this randomized consensus [57], the processes "binary value broadcast"—or "BV-broadcast" for short—their input binary value. The binary value broadcast (detailed later in Sect. 3.1) simply consists of broadcasting (including to oneself) a value, then rebroadcasting (or *echoing*) any value received from $t + 1$ distinct processes and finally bv-delivering any value received from $2t + 1$ distinct processes. These delivered values are then broadcast to the other processes and all correct processes record, into the set *values*, the values received from $n - t$ distinct processes that are among the ones previously delivered. For any correct process p, if *values* happen to contain only the value c returned by the common coin then p decides this value, if *values* contains only the other binary value $\neg c$, then p sets its estimate to this value and if *values* contains two values, then p sets its estimate to c. Then p moves to the next round until it decides.

[1] https://forum.poa.network/t/posdao-white-paper/2208.

Liveness issue. The problem is that in practice, as the communication is asynchronous, the common coin cannot return at the exact same time at all processes. In particular, if some correct processes are still at the beginning of their round r while the adversary observes the outcome of the common coin for round r then the adversary can prevent progress among the correct processes by controlling messages between correct processes and by sending specific values to them. Even if a correct process invokes the common coin before the Byzantine process, then the Byzantine can prevent correct processes from progressing.

Counter-example. To illustrate the issue, we consider a simple counter-example with $n = 4$ processes and $t = 1$ Byzantine process. Let p_1, p_2, and p_3 be correct processes with input values 0, 1, 1, respectively, and let p_4 be a Byzantine process. The goal is for process p_4 to force some correct processes to deliver $\{0, 1\}$ and another correct process to deliver $\{\neg c\}$ where c is the value returned by the common coin in the current round. As the Byzantine process has control over the network, it prevents p_2 from receiving anything before guaranteeing that p_1 and p_3 deliver $\{0, 1\}$. It is easy to see that p_4 can force p_1 and p_3 to bv-deliver 1 so let us see how p_4 forces p_1 and p_3 to deliver 0. Process p_4 sends 0 to p_3 so that p_3 receives value 0 from both p_1 and p_4, and thus echoes 0. Then p_4 sends 0 to p_1. Process p_1 then receives value 0 from p_3, p_4 and itself, hence p_1 echoes and delivers 0. Similarly, p_3 receives value 0 from p_1, p_4 and itself, hence p_3 delivers 0. To conclude p_1 and p_3 deliver $\{0, 1\}$. Processes p_1, p_3, and p_4 invoke the coin and there are two cases to consider depending on the value returned by the coin c.

- **Case** $c = 0$: Process p_2 receives now 1 from p_3, p_4 and itself, so it delivers 1.
- **Case** $c = 1$: This is the most interesting case, as p_4 should prevent some correct process, say p_2, from delivering 1 even though 1 is the most represented input value among correct processes. Process p_4 sends 0 to p_2 and p_3 so that both p_2 and p_3 receive value 0 from p_1 and p_4 and thus both echo 0. Due to p_3's echo, p_2 receives $2t + 1$ 0s and p_2 delivers 0.

At least two correct processes obtain $values = \{0, 1\}$ and another correct process can obtain $values = \{\neg c\}$. It follows that the correct processes with $values = \{0, 1\}$ adopt c as their new estimate while the correct process with $values = \{\neg c\}$ takes $\neg c$ as its new estimate and no progress can be made within this round. Finally, if the adversary (controlling p_4 in this example) keeps this strategy, then it will produce an infinite execution without termination.

Alternative and counter-measure. The problem would be fixed if we could ensure that the common coin always returns at the correct processes before returning at a Byzantine process; however, we cannot distinguish a correct process from a Byzantine process that acted correctly. We are thankful to the authors of the randomized algorithm for confirming our counter-example, they also wrote a remark in [58] indicating that both a fair scheduler and a perfect common coin were actually needed for the consensus of [57] to converge with high probability; however, no counter-example motivating the need for a fair scheduler was proposed. The intuition behind the fair scheduler is that it requires to have the same probability of receiving messages

in any order [15] and thus limits the power of the adversary on the network. A new algorithm [58] does not suffer from the same problem and offers the same asymptotic complexity in message and time as [57] but requires more communication steps, it could be used as an alternative randomized consensus in HoneyBadger to cope with this issue. Cachin and Zanolini [21] detailed recently the aforementioned counter-example and proposed a fix to [57] that retains its simplicity. Finally, a similar bug report to the aforementioned counter-example was also reported by Ethan MacBrough[2] who proposes a patch but we are unaware of any proof.

2.2 The Ethereum Blockchain and Its Upcoming Casper Consensus

Casper [18, 80] is an alternative to the existing longest branch technique to agree on a common block within Ethereum. It is well known that Ethereum can experience disagreement when different processes receive distinct blocks for the same index. These disagreements are typically resolved by waiting until the longest branch is unanimously identified. Casper aims at solving this issue by offering consensus.

The Casper FFG consensus algorithm. The FFG variant of Casper is intended to be integrated to Ethereum v2.0 during phase 0 [38]. It is claimed to ensure finality [18], a property that may seem, at first glance, to result from the termination of consensus. The model of Casper assumes authentication, synchrony and that strictly less than $1/3$ stake is owned by Byzantine processes. Casper builds a "blockchain tree" consisting of a partially ordered set of blocks. The genesis block as well as blocks at indices multiple of 100 are called *checkpoints*. Validator processes vote for a link between checkpoints of a common branch and a checkpoint is *justified* if it is the initial, so-called *genesis*, block, or there is a link from a justified checkpoint pointing to it voted by a supermajority of $\lfloor \frac{2n}{3} \rfloor + 1$ validators.

Liveness issue. Note first that Casper executes speculatively and that there is not a single consensus instance per level of the Casper blockchain tree. Each time an agreement attempt at some level of the tree fails due to the lack of votes for the same checkpoint, the height of the tree grows. Unfortunately, it has been observed that nothing guarantees the termination of Casper FFG [28] and we present below an example of infinite execution.

Counter-example. To illustrate why the consensus does not terminate in this model, let h be the level of the highest block that is justified.

1. Validators try to agree on a block at level $h + k$ $(k > 0)$ by trying to gather $\lfloor \frac{2n}{3} \rfloor + 1$ votes for the same block at level $h + k$ (or more precisely the same link from level h to $h + k$). This may fail if, for example, $\frac{n}{3}$ validators vote for one of three distinct blocks at this level $h + k$.

[2] https://github.com/amiller/HoneyBadgerBFT/issues/59.

2. Upon failure to reach consensus at level $h + k$, the correct validators, who have voted for some link from height h to $h + k$ and are incentivized to abstain from voting on another link from h to $h + k$, can now try to agree on a block at level $h + k'$ ($k' > k$), but again no termination is guaranteed.

The same steps (1) and (2) may repeat infinitely often. Note that plausible liveness [18, Theorem 2] is still fulfilled in that the supermajority "can" always be produced as long as you have infinite memory, but no such supermajority link is ever produced in this infinite execution.

Alternative and counter-measure. Another version of Casper, called CBC, has also been proposed [80]. It is claimed to be "correct by construction", hence the name CBC. This could potentially be used as a replacement to FFG Casper for Ethereum v2.0 even in phase 0 for applications that require consensus, and thus termination.

2.3 Known Problems in Blockchain Byzantine Consensus Algorithms

To show that our two counter-examples presented above are not isolated cases in the context of blockchains, we also list below four counter-examples from the literature that were reported by colleagues and affect the Ripple consensus algorithm, Tendermint and Zyzzyva. This adds to the severity of the problem of proving algorithm by hand before using them in critical applications like blockchains.

The XRP ledger and the quorums of the Ripple consensus. The Ripple consensus [69] is a consensus algorithm originally intended to be used in the blockchain system developed by the company Ripple. The algorithm is presented at a high level as an algorithm that uses unique node lists as a set of *quorums* or mutually intersecting sets that each individual process must contact to guarantee that its request will be stored by the system or that it can retrieve consistent information about asset ownership. The original but deprecated white paper [69] assumed that quorums overlap by about 20%.

Later, some researchers published an article [7] indicating that the algorithm was inconsistent and listing the environmental conditions under which consensus would not be solved and its safety would be violated. They offered a fix in order to remedy this inconsistency through the use of different assumptions, requiring that quorums overlap by strictly more than 40%. Finally, the Ripple consensus algorithm has been replaced by the XRP ledger consensus protocol [24] called ABC-Censorship-Resilience under synchrony in part to fix this problem.

The Tendermint blockchain and its locking variant to PBFT. Tendermint [49] has similar phases as PBFT [22] and works with asynchronous rounds [35]. In each round, processes propose values in turn (phase 1), the proposed value is prevoted (phase 2), precommitted when prevoted by sufficiently many[3] processes (phase 3)

[3] "Sufficiently many" processes stand for at least $\lfloor \frac{2n}{3} \rfloor + 1$ among n processes.

and decided when precommitted by sufficiently many processes. To progress despite failures, processes stay in a phase only for up to a timeout period. A difference with PBFT is that a correct process produces a proof-of-lock of v at round r if it precommits v at round r. A correct process can only prevote v' if it did not precommit a conflicting value $v \neq v'$.

As we restate here, there exists a counter-example [5] that illustrates the safety issue with four processes p_1, p_2, p_3, and p_4 among which p_4 is Byzantine that propose in the round of their index number. In the first round, correct processes prevote v, p_1, and p_2 lock v in this round and precommit it, p_1 decides v while p_2 and p_3 do not decide, before p_1 becomes slow. In the second round, process p_4 informs p_3 that it prevotes v so that p_3 prevotes, precommits, and locks v in round 2. In the third round, p_3 proposes v locked in round 2, forcing p_2 to unlock v and in the fourth round, p_4 forces p_3 to unlock v in a similar way. Finally, p_1 does not propose anything and p_2 proposes another value $v' \neq v$ that gets decided by all. It follows that correct processes p_1 and p_2 decide differently, which violates agreement. Since this discovery, Tendermint kept evolving and the authors of the counter-example acknowledged that some of the issues they reported were fixed [6], the authors also informed us that they notified the developers but ignore whether this particular safety issue has been fixed.

Zyzzyva and the SBFT concurrent fast and regular paths. Zyzzyva [48] is a Byzantine consensus that requires view-change and combines a fast path where a client can learn the outcome of the consensus in three message delays and a regular path where the client needs to collect a commit-certificate with $2f + 1$ responses where f is the actual number of Byzantine faults. The same optimization is currently implemented in the SBFT permissioned blockchain [39] to speed up termination when all participants are correct and the communication is synchronous.

There exist counter-examples [1] that illustrate how the safety property of Zyzzyva can be violated. The idea of one counter-example consists of creating a commit-certificate for a value v, then experiencing a first view-change (due to delayed messages) and deciding another value v' for a given index before finally experiencing a second view-change that leads to undoing the former decision v' but instead deciding v at the same index. SBFT is likely to be immune to this issue as the counter-example was identified by some of the authors of SBFT. But a simple way to cope with this issue is to prevent the two paths from running concurrently as in the simpler variant of Zyzzyva called Azyzzva [8].

The Quorum blockchain and its IBFT consensus. IBFT [52] is a Byzantine fault-tolerant consensus algorithm at the heart of the Quorum blockchain designed by JP Morgan. It is similar to PBFT [22] except that it offers a simplified version of the PBFT view-change by getting rid of new-view messages. It aims at solving consensus under partial synchrony. The protocol assumes that no more than $t < n/3$ processes—usually referred by IBFT as "validators"—are Byzantine.

As reported in [68], IBFT does not terminate in a partially synchronous network even when failures are crashes. More precisely, IBFT cannot guarantee that if at least one honest validator is eventually able to produce a valid finalized block then the

transaction it contains will eventually be added to the local transaction ledger of any other correct process. IBFT v2.x [68] fixes this problem but requires a transaction to be submitted to all correct validators for this transaction to be eventually included in the distributed permissioned transaction ledger. The proof was made by hand and we are not aware of any automated proof of this protocol as of today.

3 A Methodology for Verifying Blockchain Components

In this section, we explain how we verified the binary value broadcast blockchain component using the Byzantine model checker. Then we explain how this helped us verify the consistency of a slight variant of the binary consensus of DBFT used in Red Belly Blockchain under the round-rigid adversary assumption. Note that the DBFT binary consensus algorithm was since then proven safe and live without this assumption [11].

3.1 Preliminaries on ByMC and BV-Broadcast

Byzantine model checker. Fault-tolerant distributed algorithms, like the Byzantine fault-tolerant broadcast primitive presented below, are often based on parameters, like the number n of processes, the maximum number of Byzantine faults t, or the number of Byzantine faults f. Threshold-guarded algorithms [45, 46] use these parameters to define threshold-based guard conditions that enable transitions to different states. Once a correct process receives a number of messages that reaches the threshold, it progresses by taking some transition to a new state. To circumvent the undecidability of model checking on infinite systems, Konnov, Schmid, Veith, and Widder introduce two parametric interval abstractions [44] that model *(i)* each process with a finite-state machine independent of the parameters and *(ii)* the whole system with abstract counters that quantify the number of processes in each state in order to obtain a finite-state system. Finally, they group a potentially infinite number of runs into an execution schema in order to allow bounded model checking, based on an SMT solver, over all the possible execution schemas [46]. ByMC [47] verifies threshold automata with this model checking and has been used to prove various distributed algorithms, like atomic commit or reliable broadcast. Given a set of safety and liveness properties, it outputs traces showing that the properties are satisfied in all the reachable states of the threshold automaton. Until 2018, correctness properties were only verified on one round but more recently the threshold automata framework was extended to randomized algorithms, making possible to verify algorithms such as Ben-Or's randomized consensus under round-rigid adversaries [13].

Binary value broadcast. The binary value broadcast [57], also denoted BV-broadcast, is a Byzantine fault-tolerant communication abstraction used in blockchains [31, 56] that works in an asynchronous network with reliable channels

where the maximum number of Byzantine failures is $t < n/3$. The BV-broadcast guarantees that no values broadcasted exclusively by Byzantine processes can be delivered by correct processes. This helps limiting the power of the adversary to make sure that a Byzantine consensus algorithm converges toward a value. In particular, by requiring that all correct processes BV-broadcast their proposals, one can guarantee that all correct processes will eventually observe their proposals, regardless of the values proposed by Byzantine processes. The binary value broadcast finds applications in blockchains: first, it is implemented in HoneyBadger [56] to detect that correct processes have proposed diverging values in order to toss a common coin that returns the same result across distributed correct processes, to make them converge to a common decision. Second, Red Belly Blockchain [31] and the accountable blockchain that is derived from it [26, 27] implement the BV-broadcast to detect whether the protocol can converge toward the parity of the round number by simply checking that it corresponds to one of the values that were "bv-delivered".

The BV-broadcast abstraction satisfies the four following properties:

1. BV-Obligation. If at least $(t + 1)$ correct processes BV-broadcast the same value v, v is eventually added to the set $conts_i$ of each correct process p_i.
2. BV-Justification. If p_i is correct and $v \in conts_i$, v has been BV-broadcast by some correct process. (Identification following from receiving more than t 0s or 1s.)
3. BV-Uniformity. If a value v is added to the set $conts_i$ of a correct process p_i, eventually $v \in conts_j$ at every correct process p_j.
4. BV-Termination. Eventually the set $conts_i$ of each correct process p_i is not empty.

3.2 *Automated Verification of a Blockchain Byzantine Broadcast*

In this section, we describe how we used threshold automaton to specify the binary value broadcast algorithm and ByMC in order to verify the protocol automatically. We recall the BV-broadcast algorithm as depicted in Algorithm 1. The algorithm consists of having at least $n - t$ correct processes broadcasting a binary value. Once a correct process receives a value from $t + 1$ distinct processes, it broadcasts it if it did not do it already. Once a correct process receives a value from $2t + 1$ distinct processes, it delivers it. Here the delivery is modeled by adding the value to the set

Algorithm 1 The binary value broadcast algorithm

```
1: bv-broadcast(MSG, val, conts, i): // bv-broadcast filters out values proposed only by Byzantine
2:   broadcast(BV, ⟨val, i⟩) // broadcast binary value val
3:   repeat: // re-broadcast a received value only if it is sufficiently represented
4:     if (BV, ⟨v, ∗⟩) received from (t + 1) distinct processes but not yet broadcast then
5:       broadcast(BV, ⟨v, i⟩) // echo v
6:     if (BV, ⟨v, ∗⟩) received from (2t + 1) distinct processes then // from correct majority
7:       conts ← conts ∪ {v} // deliver v
```

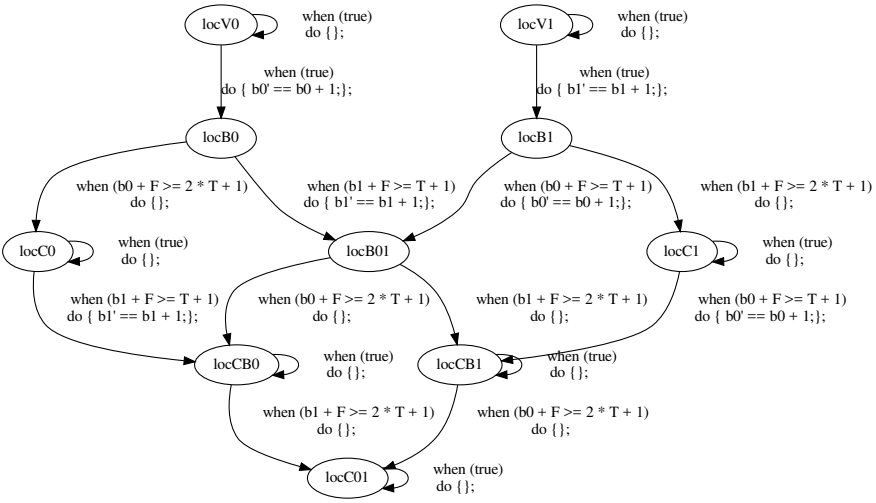

Fig. 1 The threshold automaton of the binary value broadcast algorithm

conts, which will simplify the description of our slight variant of the DBFT binary consensus algorithm in Sect. 4.

Specifying the distributed algorithm in a threshold automaton. Let us describe how we specify Algorithm 1 as a threshold automaton depicted in Fig. 1. Each state of the automaton or node in the corresponding graph represents a local state of a process. A process can move from one state to another thanks to an edge, called a *rule*. A rule has the form $\phi \mapsto u$, where ϕ is a guard and u an action on the shared variables. When the guard evaluates to true (e.g., more than $t + 1$ messages of a certain type have been sent), the action is executed (e.g., the shared variable s is incremented).

In Algorithm 1, we can see that only two types of messages are exchanged: process i can only send either $(\mathsf{BV}, \langle 0, i\rangle)$ or $(\mathsf{BV}, \langle 1, i\rangle)$. Each time a value is sent by a correct process, it is actually broadcasted to all processes. Thus, we only need two shared variables $b0$ and $b1$ corresponding to the value 0 and 1 in the automaton (cf. Fig. 1). Incrementing $b0$ is equivalent to broadcasting $(\mathsf{BV}, \langle 0, i\rangle)$. Initially, each correct process immediately broadcasts its value. This is why the guard for the first rule is true: a process in $locV0$ can immediately move to $locB0$ and send 0 during the transition.

We then enter the *repeat* loop of the pseudocode. The two *if* statements are easily understandable as threshold guards. If more than $t + 1$ messages with value 1 are *received*, then the process should broadcast 1 (i.e., increment $b1$) since it has not already been done. Interestingly, the corresponding guard is $b1 + f \geq t + 1$. Indeed, the shared variable $b1$ only counts the messages *sent* by correct processes. However, the f faulty processes might send messages with arbitrary values. We want to consider all the possible executions, so the earliest moment a correct process can move from $locB0$ to $locB01$ is when the f faulty processes and $t + 1 - f$ correct

processes have sent 1. The other edge leaving $locB0$ corresponds to the second *if* statement, that is, satisfied when $2t + 1$ messages with value 0 have been received. In state $locC0$, the value 0 has been delivered. A process might stay in this state forever, so we add a self-loop with guard condition set to *true*.

After the state $locC0$, a process is still able to broadcast 1 and eventually deliver 1 after that. After the state $locB01$, 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, we remark that the automaton is a directed acyclic graph. On every path of the graph, we can verify that a shared variable is incremented only once. This is because in the pseudocode, a value can be broadcasted only if it has not been broadcasted before.

Finally, the states of the automaton correspond to the following (unique) situations for a correct process:

- **locV0**. Initial state with value 0, nothing has been broadcasted nor delivered.
- **locV1**. Initial state with value 1, nothing has been broadcasted nor delivered.
- **locB0**. Only 0 has been broadcasted, nothing has been delivered.
- **locB1**. Only 1 has been broadcasted, nothing has been delivered.
- **locB01**. Both 0 and 1 have been broadcasted, nothing has been delivered.
- **locC0**. Only 0 has been broadcasted, only 0 has been delivered.
- **locCB0**. Both 0 and 1 have been broadcast, only 0 has been delivered.
- **locC1**. Only 1 has been broadcasted, only 1 has been delivered.
- **locCB1**. Both 0 and 1 have been broadcasted, only 1 has been delivered.
- **locC01**. Both 0 and 1 have been broadcasted, both 0 and 1 have been delivered.

Once the pseudocode is converted into a threshold automaton depicted in Fig. 1, one can simply write the corresponding specification in the threshold automata language to obtain the specification listed below (Listing 1) for completeness.

Defining the correctness properties and fairness assumptions. The above automaton is only the first half of the verification work. The second half consists in specifying the correctness properties that we would like to verify on the algorithm. We use temporal logic on the algorithm variables (number of processes in each location, number of messages sent, and parameters) to formalize the properties. In the case of the BV-broadcast, the BV-Justification property of the BV-broadcast is "If p_i is correct and $v \in cons_i$, v has been BV-broadcast by some correct process". Given $\Diamond$, $\rightarrow$ and $||$ with the LTL semantics of "eventually", "implies", and "or", respectively, we translate this property in the following conjunction:

$$\begin{cases} justification0 : (\Diamond(locC0 \neq 0 \,||\, locC01 \neq 0)) \rightarrow \\ \qquad\qquad\qquad\qquad (locV0 \neq 0), \\ justification1 : (\Diamond(locC1 \neq 0 \,||\, locC01 \neq 0)) \rightarrow \\ \qquad\qquad\qquad\qquad (locV1 \neq 0). \end{cases}$$

Liveness properties are longer to specify, because we need to take into account some fairness constraints. Indeed, a threshold automaton describes processes evolv-

ing in an asynchronous setting without additional assumptions. An execution in which a process stays in a state forever is a valid execution, but it does not make any progress. If we want to verify some liveness properties, we have to add some assumptions in the specification. For instance, we require that processes eventually leave the states of the automaton as long as they have received enough messages to enable the condition guarding the outgoing rule. In other words, a liveness property will be specified as follows: *liveness_property* : *fairness_condition* → *property*.

Note that this assumption is natural and differs from the round-rigidity assumption that requires the adversary to eventually take any applicable transition of an infinite execution. Finally, we wrote a threshold automaton specification whose `.ta` file is presented in Listing 1 in only 116 lines.

Experimental results. On a simple laptop with an Intel Core i5-7200U CPU running at 2.50GHz, verifying all the correctness properties for BV-broadcast takes less than 40 s. For simple properties on well-specified algorithms, such as the ones of the benchmarks included with ByMC, the verification time can be less than one second. This result encouraged us to verify a complete Byzantine consensus algorithm in Sect. 4 that builds upon the binary value broadcast.

Debugging the manual conversion of the algorithm to the automaton. It is common that the specification does not hold at first try, because of some mistakes in the threshold automaton model or in the translation of the correctness property into a formal specification. In such cases, ByMC provides a detailed output and a counter-example showing where the property has been violated. We reproduced such a counter-example in Fig. 2 with an older preliminary version of our specification. This specification was wrong because a liveness property did not hold. ByMC gave parameters and provided an execution ending with a loop, such that the condition of the liveness was never met. This trace helped us understand the problem in our specification and allowed us to fix it to obtain the correct specification we

```
1  N:=34; T:=11; F:=1;
2  0 (F 0) x 0: b0:=0; b1:=0; K[pc:0]:=21; K[pc:1]:=12; K[*]:=0;
3  1 (F 1) x 1: b0:=1; K[pc:0]:=20; K[pc:2]:=1;
4
5                              (...)
6
7  24 (F 52) x 1:  b1:=21; K[pc:5]:=12; K[pc:7]:=21;
8  ****************
9  b0:=33; b1:=21; K[pc:0]:=0; K[pc:1]:=0; K[pc:2]:=0;
10 K[pc:3]:=0; K[pc:4]:=0; K[pc:5]:=12; K[pc:6]:=0; K[pc:7]:=21;
11 K[pc:8]:=0; K[pc:9]:=0;
12
13 ****** LOOP *******
14 N:=34; T:=11; F:=1;
15 25 (F 83) x 1:  <self-loop>
16 ****************
17 K[pc:2]:=0; K[pc:4]:=0; K[pc:5]:=12; K[pc:7]:=21; K[pc:8]:=0;
18 K[pc:9]:=0;
```

Fig. 2 Truncated counter-example produced by ByMC for a faulty specification of BV-broadcast

illustrated before in Fig. 1. Building upon this successful result, we specified a more complex Byzantine consensus algorithm that uses the same broadcast abstraction but we did not encounter any bug during this process and our first specification was proved correct by ByMC. The pseudocode, threshold automaton specification, and experimental results are presented in Sect. 4.

Listing 1 Threshold automaton specification for the binary value broadcast communication primitive

```
thresholdAutomaton Proc {
 local pc; shared b0, b1;
 parameters N, T, F;

 assumptions (0) { N>3*T; T>=F; T>=1; }

 locations (0) {
  locV0:[0]; locV1:[1]; locB0:[2];
  locB1:[3]; locB01:[4]; locC0:[5];
  locC1:[6]; locCB0:[7];
  locCB1:[8]; locC01:[9];
 }

 inits (0) {
  (locV0+locV1)==N-F;
  locB0==0; locB1==0; locB01==0;
  locC0==0; locC1==0; locCB0==0;
  locCB1==0; locC01==0; b0==0; b1==0;
 }

 rules (0) {
% for v in [0, 1]:
 1: locV${v} -> locB${v}
   when (true)
   do { b${v}'==b${v}+1;
      unchanged(b${1-v}); };

 2: locB${v} -> locB01
   when (b${1-v}+F>=T+1)
   do { b${1-v}'==b${1-v}+1;
      unchanged(b${v}); };

 3: locB${v} -> locC${v}
   when (b${v}+F>=2*T+1)
   do { unchanged(b0, b1); };

 2: locC${v} -> locCB${v}
   when (b${1-v}+F>=T+1)
   do { b${1-v}'==b${1-v}+1;
      unchanged(b${v}); };

 3: locB01 -> locCB${v}
   when (b${v}+F>=2*T+1)
   do { unchanged(b0, b1); };

 3: locCB${v} -> locC01
   when (b${1-v}+F>=2*T+1)
   do { unchanged(b0, b1); };

 /* self loops */
 10: locV${v} -> locV${v}
   when (true) do {unchanged(b0, b1);};

 10: locC${v} -> locC${v}
   when (true) do {unchanged(b0, b1);};

 10: locCB${v} -> locCB${v}
   when (true) do {unchanged(b0, b1);};
% endfor

 10: locC01 -> locC01
   when (true) do {unchanged(b0, b1);};
 }

 specifications (0) {

 % for v in [0,1]:
  obligation${v}:
<>[]((locV0==0) && (locV1==0) &&
(locB0==0 || b1<T+1) && (locB1==0 || b0<T+1) &&
(locB0==0 || b0<2*T+1) && (locB1==0 || b1<2*T+1) &&
(locB01==0 || b0<2*T+1) && (locB01==0 || b1<2*T+1) &&
(locC0==0 || b1<T+1) && (locC1==0 || b0<T+1) &&
(locCB0==0 || b1<2*T+1) && (locCB1==0 || b0<2*T+1))
->
    ((locV${v}>=T+1)
->
<>(locV0==0 && locV1==0 &&
      locB0==0 && locB1==0 &&
      locB01==0 && locC${1-v}==0 &&
      locCB${1-v}==0));

  justification${v}: (<>(locC${v}!=0
|| locCB${v}!=0 || locC01!=0))
-> (locV${v}!=0);

 uniformity${v}:
<>[]((locV0==0) && (locV1==0) &&
(locB0==0 || b1<T+1) && (locB1==0 || b0<T+1) &&
(locB0==0 || b0<2*T+1) && (locB1==0 || b1<2*T+1) &&
(locB01==0 || b0<2*T+1) && (locB01==0 || b1<2*T+1) &&
(locC0==0 || b1<T+1) && (locC1==0 || b0<T+1) &&
(locCB0==0 || b1<2*T+1) && (locCB1==0 || b0<2*T+1))
->
 (<>(locC${v}!=0 || locCB${v}!=0 || locC01!=0)
->
<>[](locC${1-v}==0 && locCB${1-v}==0));
 % endfor

 termination:
<>[]((locV0==0) && (locV1==0) &&
     (locB0==0 || b1<T+1) &&
     (locB1==0 || b0<T+1) &&
     (locB0==0 || b0<2*T+1) &&
     (locB1==0 || b1<2*T+1) &&
     (locB01==0 || b0<2*T+1) &&
     (locB01==0 || b1<2*T+1) &&
     (locC0==0 || b1<T+1) &&
     (locC1==0 || b0<T+1) &&
     (locCB0==0 || b1<2*T+1) &&
     (locCB1==0 || b0<2*T+1))
->
<>(locV0 ==0 && locV1 ==0 &&
      locB0 ==0 && locB01==0);
  }
} /* Proc */
```

4 Verifying a Blockchain Byzantine Consensus Algorithm

The Democratic Byzantine Fault-Tolerant consensus algorithm [30] is a Byzantine consensus algorithm that does not require a leader. It was implemented in the recent Red Belly Blockchain [32] to offer high performance through multiple proposers and was used in Polygraph [26, 27] to detect malicious participants responsible of disagreements when $t \geq n/3$ and in the Long-Lasting Blockchain [67] to recover from forks by excluding misbehaving participants. As depicted in Algorithm 2, a slight variant of its binary consensus, made simpler than the original algorithm by omitting timeouts, proceeds in asynchronous rounds that correspond to the iterations of a loop where correct processes refine their estimate value.

Algorithm 2 A variant of the DBFT binary Byzantine consensus algorithm

Notation: "Received k messages" is a shortcut for "Received k messages from different processes in the same round r as the current round."

```
1: propose(v):
2:   est ← v // initial estimate is the proposed value
3:   r ← 0 // initialize the round number
4:   repeat: // repeat in asynchronous rounds
5:     r ← r + 1; // increment the round number
6:     broadcast(tag = BV, round = r, value = est) // initial broadcast
7:     while true do // start of binary value broadcast phase
8:       if received (t + 1) BV messages with value w and w not broadcast yet then
9:         broadcast(tag = BV, round = r, value = w) // rebroadcast legitimate estimates
10:      if received (2t + 1) BV messages with value w then // recvd from correct majority
11:        broadcast(tag = ECHO, round = r, value = w) // broadcast ECHO message
12:        break // exit the while loop to proceed to next phase
13:    while true do // wait to have received enough messages
14:      echoes ← {w ∈ {0, 1} : received (2t + 1) BV messages with value w}
15:      if received (n − t) ECHO messages with value w ∈ echoes then
16:        est ← w // refine estimate
17:        if w = r mod 2 and not decided yet then // depending on the singleton value w...
18:          decide(w) // ...decide the parity of the round
19:        break // exit the while loop to proceed to next round
20:      if received (n − t) ECHO messages and echoes = {0, 1} then // all bv-delivered
21:        est ← r mod 2 // set estimate to round parity
22:        break // exit the while loop to proceed to next round
23:    if decided in round r_i − 2 then exit // exit the consensus only after having helped others
```

Initially, each correct process sets its estimate to its input value. Correct processes broadcast these estimates and rebroadcast only values received by $t + 1$ distinct processes because they are proposed by correct processes. Each value received from $2t + 1$ distinct processes (and from a majority of correct processes) is stored in the *echoes* set and is broadcasted as part of an ECHO message. The ECHO value received from $n - t$ distinct processes that also belongs to *echoes* becomes the new estimate (line 16) for the next round. If this value corresponds to the parity of the

round, then the correct process decides this value. If *echoes* contain both values, then the estimate for the next round becomes the parity of the round. As opposed to the original and partially synchronous deterministic version [30], this variant uses one less broadcast phase and offers termination in an asynchronous network under round-rigidity that requires the adversary to eventually perform any applicable transition within an infinite execution. This assumption was previously used to show termination of another algorithm with high probability [13]. The specification of our consensus algorithm in threshold automata is depicted in Listing 2.

Listing 2 Variant of the DBFT binary Byzantine consensus

```
thresholdAutomaton Proc {

  local pc;

  /* Messages sent by correct proc. */
  /* First round */
  shared b0, b1;
  shared e0, e1;
  /* Second round */
  shared b0x, b1x;
  shared e0x, e1x;

  parameters N, T, F;

  assumptions (0) {
    N > 3 * T;
    T >= F;
    T >= 1;
  }

  locations (0) {
    locV0:    [0];
    locV1:    [1];
    locB0:    [2];
    locB1:    [3];
    locB01:   [4];
    locC:     [5];
    locE0:    [6];
    locE1:    [7];
    locD1:    [8];
    locB0x:   [9];
    locB1x:   [10];
    locB01x:  [11];
    locCx:    [12];
    locE0x:   [13];
    locE1x:   [14];
    locD0:    [15];
  }

  inits (0) {
    (locV0 + locV1) == N - F;

    locB0 == 0;
    locB1 == 0;
    locB01 == 0;
    locC == 0;
    locE0 == 0;
    locE1 == 0;
    locD1 == 0;
    locB0x == 0;


  rules (0) {
% for v in [0, 1]:
  1: locV${v} -> locB${v}
       when (true)
       do { b${v}' == b${v} + 1;
            unchanged(b${1-v}, e0, e1);
            unchanged(b0x, b1x, e0x, e1x);
          };
% endfor

% for v in [0, 1]:
  2: locB${v} -> locB01
       when (b${1-v} + F >= T + 1)
       do { b${1-v}' == b${1-v} + 1;
            unchanged(b${v}, e0, e1);
            unchanged(b0x, b1x, e0x, e1x);
          };
% endfor

% for v in [0, 1]:
  3: locB${v} -> locC
       when (b${v} + F >= 2 * T + 1)
       do { e${v}' == e${v} + 1;
            unchanged(b0, b1, e${1-v});
            unchanged(b0x, b1x, e0x, e1x);
          };
% endfor

% for v in [0, 1]:
  4: locB01 -> locC
       when (b${v} + F >= 2 * T + 1)
       do { e${v}' == e${v} + 1;
            unchanged(b0, b1, e${1-v});
            unchanged(b0x, b1x, e0x, e1x);
          };
% endfor

  5: locC -> locD1
       when (e1 + F >= N - T
&& b1 + F >= 2 * T + 1)
       do {
            unchanged(b0, b1, e0, e1);
            unchanged(b0x, b1x, e0x, e1x);
       };

  6: locC -> locE0
       when (e0 + F >= N - T
&& b0 + F >= 2 * T + 1)
       do {
            unchanged(b0, b1, e0, e1);
            unchanged(b0x, b1x, e0x, e1x);
       };

```

```
      locB1x == 0;
      locB01x == 0;
      locCx == 0;
      locE0x == 0;
      locE1x == 0;
      locD0 == 0;

      b0 == 0;
      b1 == 0;
      e0 == 0;
      e1 == 0;
      b0x == 0;
      b1x == 0;
      e0x == 0;
      e1x == 0;
    }
```

```

% for v in [0, 1]:
  9: locB${v}x -> locB01x
       when (b${1-v}x + F >= T + 1)
       do { b${1-v}x' == b${1-v}x + 1;
            unchanged(b0, b1, e0, e1);
            unchanged(b${v}x, e0x, e1x);
          };
% endfor

% for v in [0, 1]:
  10: locB${v}x -> locCx
       when (b${v}x + F >= 2 * T + 1)
       do { e${v}x' == e${v}x + 1;
            unchanged(b0, b1, e0, e1);
            unchanged(b0x, b1x, e${1-v}x);
          };
% endfor

% for v in [0, 1]:
  11: locB01x -> locCx
       when (b${v}x + F >= 2 * T + 1)
       do { e${v}x' == e${v}x + 1;
            unchanged(b0, b1, e0, e1);
            unchanged(b0x, b1x, e${1-v}x);
          };
% endfor

  12: locCx -> locD0
       when (e0x + F >= N - T
&& b0x + F >= 2 * T + 1)
       do {
            unchanged(b0, b1, e0, e1);
            unchanged(b0x, b1x, e0x, e1x);
       };

  13: locCx -> locE1x
       when (e1x + F >= N - T
&& b1x + F >= 2 * T + 1)
       do {
            unchanged(b0, b1, e0, e1);
            unchanged(b0x, b1x, e0x, e1x);
       };


  14: locCx -> locE0x
       when (e0x + e1x + F >= N - T
&& b0x + F >= 2 * T + 1
&& b1x + F >= 2 * T + 1)
       do {
            unchanged(b0, b1, e0, e1);
            unchanged(b0x, b1x, e0x, e1x);
       };


  /* self loops */

% for v in [0, 1]:
  10: locV${v} -> locV${v}
       when (true)
```

```
    7: locC -> locE1
         when (e0 + e1 + F >= N - T
  && b0 + F >= 2 * T + 1
  && b1 + F >= 2 * T + 1)
         do {
              unchanged(b0, b1, e0, e1);
              unchanged(b0x, b1x, e0x, e1x);
         };

  % for v in [0, 1]:
    8: locE${v} -> locB${v}x
         when (true)
         do { b${v}x' == b${v}x + 1;
              unchanged(b0, b1, e0, e1);
              unchanged(b${1-v}x, e0x, e1x);
            };
  % endfor
```

```

% for v in [0, 1]:
  10: locE${v}x -> locE${v}x
      when (true)
      do {
          unchanged(b0, b1, e0, e1);
          unchanged(b0x, b1x, e0x, e1x);
      };
% endfor
  }
  specifications (0) {

  % for v in [0, 1]:
      validity${v}:
      (locV${1-v} == 0) ->
      [](locD${1-v} == 0 && locE${1-v}x == 0);
  % endfor

  % for v in [0, 1]:
    agreement${v}:
    [](!(locD${v} != 0) ->
    [](locD${1-v} == 0 && locE${1-v}x == 0));
  % endfor

    round_termination:
<>[](
        (locV0 == 0) &&
        (locV1 == 0) &&
        (locB0
== 0 || (b1 < T + 1 && b0 < 2 * T + 1)) &&
        (locB1
== 0 || (b0 < T + 1 && b1 < 2 * T + 1)) &&
        (locB01
== 0 || (b0 < 2 * T + 1 && b1 < 2 * T + 1)) &&
        (locC == 0 ||
            ((e1 < N - T || b1 < 2 * T + 1) &&
            (e0 < N - T || b0 < 2 * T + 1) &&
            (e0 + e1 < N - T ||
                  b0 < 2 * T + 1 ||
                  b1 < 2 * T + 1) )) &&
        (locE0 == 0) &&
        (locE1 == 0) &&
        (locB0x
== 0 || (b1x < T + 1 && b0x < 2 * T + 1)) &&
        (locB1x
== 0 || (b0x < T + 1 && b1x < 2 * T + 1)) &&
        (locB01x == 0 ||
            (b0x < 2 * T + 1 && b1x < 2 * T + 1)) &&
        (locCx == 0 ||
            ((e1x < N - T || b1x < 2 * T + 1) &&
            (e0x < N - T || b0x < 2 * T + 1) &&
            (e0x + e1x < N - T ||
                  b0x < 2 * T + 1 ||
                  b1x < 2*T+1)))
    )
->
<>(
        locV0 == 0 &&
        locV1 == 0 &&
        locB0 == 0 &&
```

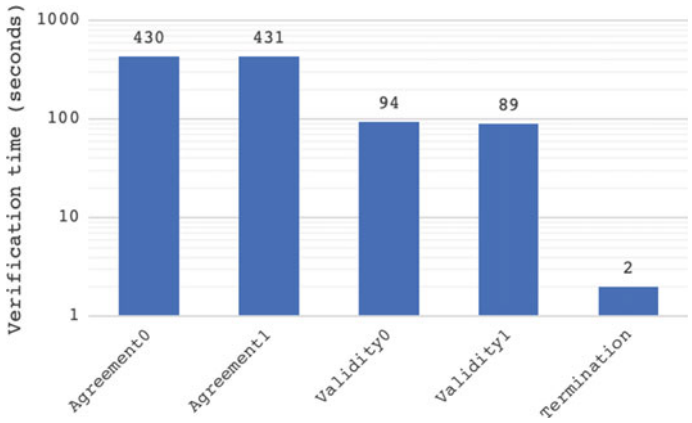

Fig. 3 Time to verify the Byzantine consensus of Algorithm 2

```
        do {
            unchanged(b0, b1, e0, e1);
            unchanged(b0x, b1x, e0x, e1x);
        };
  % endfor

  % for v in [0, 1]:
    10: locD${v} -> locD${v}
        when (true)
        do {
            unchanged(b0, b1, e0, e1);
            unchanged(b0x, b1x, e0x, e1x);
        };
  % endfor
```

```
          locB1 == 0 &&
          locB01 == 0 &&
          locC == 0 &&
          locE0 == 0 &&
          locE1 == 0 &&
          locB0x == 0 &&
          locB1x == 0 &&
          locB01x == 0 &&
          locCx == 0
        );

    }
  } /* Proc */
```

4.1 Experimental Results

The Byzantine consensus algorithm has far more states and variables than the BV-broadcast primitive and it is too complex to be verified on a personal computer. We ran the parallelized version of ByMC with MPI on a 4 AMD Opteron 6276 16-core CPU with 64 cores at 2300 MHz with 64 GB of memory. The verification times for the five properties are listed in Fig. 3 and sum up to 17 min and 26 s.

5 Related Work

The observation that some of the blockchain consensus proposals have issues is not new [20, 40]. It is now well known that the termination of existing blockchains like Ethereum requires an additional assumption like synchrony [40]. Our Ethereum counter-example differs as it considers the upcoming consensus algorithm of Ethereum v2.0. In [20], the conclusions are different from ours as they generalize on other Byzantine consensus proposals, like Tangaroa, not necessarily in use in

blockchain systems. Our focus is on consensus used in blockchains that are critical due to trading valuable assets. Note that other consistency violations related to the consensus offered in Ethereum v1.x and v2.0 have been concurrently reported [36, 37, 62].

Threshold automata already proved helpful to automate the proof of existing consensus algorithms [47]. They have even been useful in illustrating why a specification of the King-Phase algorithm [10] was incorrect [72] (due to the strictness of a lower symbol), later fixed in [14]. We did not list this as one of the inconsistency problems that affects blockchains as we are not aware of any blockchain implementation that builds upon the King-Phase algorithm. In [51], the authors use threshold guarded automata to prove two broadcast primitives and the Bosco Byzantine consensus correct; however, Bosco offers a fast path but requires another consensus algorithm for its fallback path so its correctness depends on the assumption that it relies on a correct consensus algorithm.

In general, it is hard to formally prove algorithms that work in a partially synchronous model while there exist tools to reduce the state space of synchronous consensus to finite-state model checking [4]. Part of the reason is that common partially synchronous solutions attempt to give sufficient time to processes in different asynchronous rounds by incrementing a timeout until the timeout is sufficiently large to match the unknown message delay bound. PSync [34] and ConsL [55] are languages that help reasoning formally about partially synchronous algorithms. In particular, ConsL was shown effective at verifying consensus algorithms but only for the crash fault-tolerant model. Here we used the ByMC model checker [45] for asynchronous Byzantine fault-tolerant systems and require the round-rigidity assumption to show a variant of the binary consensus of DBFT [30].

Interactive theorem provers [66, 70, 77] were used to prove consensus algorithms. In particular, the Coq proof assistant helped prove distributed algorithms [2] like two-phase commit [70], Raft [78] and the Algorand consensus algorithm [3] while Dafny [42] proved MultiPaxos. Isabelle/HOL [64] was used to prove byzantine fault-tolerant algorithms [23] and was combined with Ivy to prove the Stellar consensus protocol [53]. Theorem provers check proofs, not the algorithms. Hence, one has to invest efforts into writing detailed mechanical proofs.

In [79], the authors present TLC, a model checker for debugging a finite-state model of a TLA+ specification. TLA+ is a specification language for concurrent and reactive systems that build upon the temporal logic TLA. One limitation is that the TLA+ specification might comprise an infinite set of states for which the model checker can only give a partial proof. In order to run the TLC model checker on a TLA+ specification, it is necessary to fix the parameters such as the number of processes n or the bounds on integer values. In practice, the complexity of model checking explodes rapidly and makes it difficult to check anything beyond toy examples with a handful of processes. TLC remains useful—in particular in industry—to prove that some specifications are wrong [63]. TLA+ also comes with a proof system called TLAPS. TLAPS supports manually written hierarchically structured proofs, which are then checked by backend engines such as Isabelle, Zenon, or SMT solvers [29]. TLAPS is still being actively developed but it is already possible—albeit technical and lengthy—to prove algorithms such as Paxos (Fig. 4).

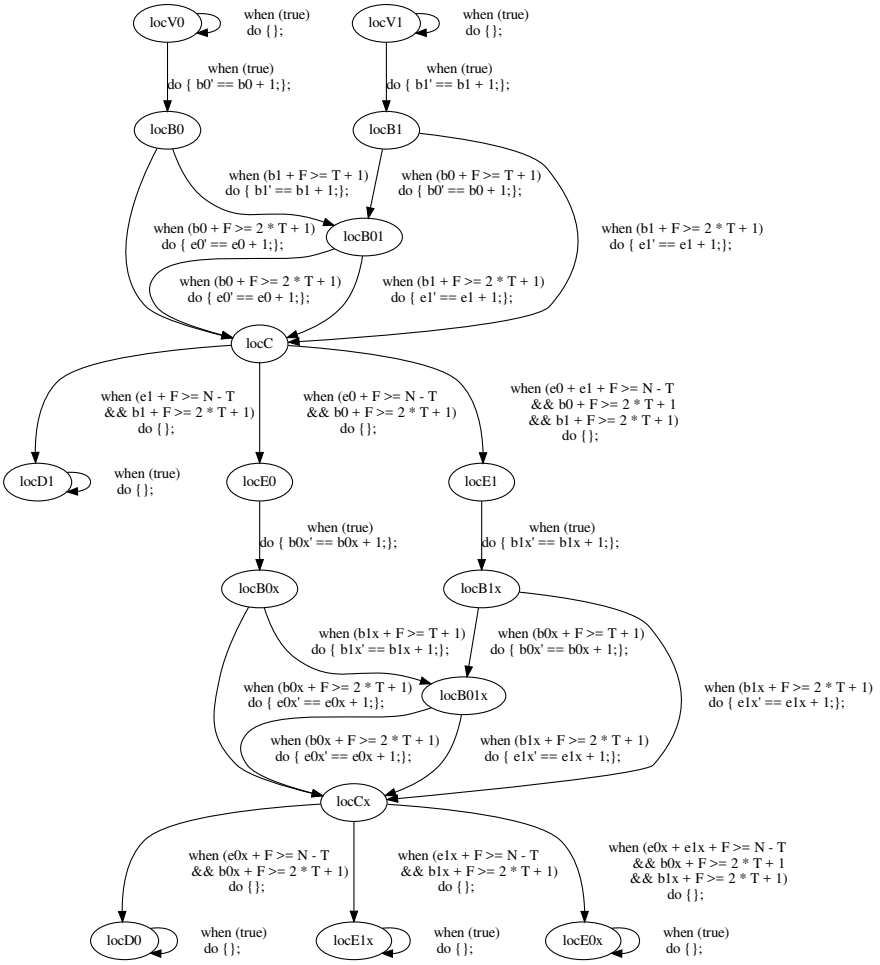

Fig. 4 The threshold automaton of the DBFT binary consensus variant

Recently, the binary consensus of DBFT [30] was formally proved safe and live using parameterized model checking [11] but without any round-rigid adversary assumption. To this end, the specification of the Byzantine consensus algorithm was split into multiple threshold automata.

6 Discussion and Conclusion

In this paper, we argued for the formal verification of blockchain Byzantine fault-tolerant algorithms as a way to reduce the numerous issues resulting from non-formal proofs for such critical applications as blockchains. In particular, we illustrated the

problem with new counter-examples of algorithms at the core of widely deployed blockchain software.

We show that it is now feasible to verify blockchain Byzantine components on modern machines thanks to the recent advances in formal verification. We illustrate it with relatively simple specifications of a broadcast abstraction common to multiple blockchains as well as a variant of the Byzantine consensus algorithm of the Red Belly Blockchain.

To verify the Byzantine consensus, we assumed a round-rigid adversary that schedules transitions in a fair way. This is not new as in [13] the model checking of the randomized algorithm from Ben-Or required a round-rigid adversary. Interestingly, we do not need this assumption to verify the binary value broadcast abstraction that works in an asynchronous model. A concomitant result replaces the round-rigidity assumption by a deterministic fairness assumption to formally verify the liveness and safety properties of the consensus algorithm of DBFT [11].

As future work, we would like to prove other Byzantine fault-tolerant algorithmic components of blockchain systems.

Acknowledgements Parts of the content of this chapter have been presented in the non-archiving workshops FRIDA'19 and ConsensusDays'21. We wish to thank Igor Konnov and Josef Widder for helping us understand the syntax and semantics of the threshold automata specification language and for confirming that ByMC verified the agreement1 property of our initial specification. We thank Tyler Crain, Achour Mostéfaoui, and Michel Raynal for discussions of the HoneyBadger counter-example, and Yackolley Amoussou-Guenou, Maria Potop-Butucaru, and Sara Tucci for discussions on the Tendermint counter-example. This research is supported under Australian Research Council Discovery Projects funding scheme (project number 180104030) entitled "Taipan: A Blockchain with Democratic Consensus and Validated Contracts" and Australian Research Council Future Fellowship funding scheme (project number 180100496) entitled "The Red Belly Blockchain: A Scalable Blockchain for Internet of Things".

References

1. Abraham, I., Gueta, G.G., Malkhi, D., Alvisi, L., Kotla, R., Martin J.-P.: Revisiting fast practical byzantine fault tolerance. Technical report (Dec 2017). arXiv
2. Altisen, K., Corbineau, P., Devismes, S.: A framework for certified self-stabilization. In: FORTE, pp. 36–51 (2016)
3. Alturki, M.A., Chen, J., Luchangco, V., Moore, B.M., Palmskog, K., Peña, L., Rosu, G.: Towards a verified model of the algorand consensus protocol in coq. In: International Workshops on Formal Methods (FM), pp. 362–367 (2019)
4. Aminof, B., Rubin, S., Stoilkovska, I., Widder, J., Zuleger F.: Parameterized model checking of synchronous distributed algorithms by abstraction. In: Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI, pp. 1–24 (2018)
5. Amoussou-Guenou, Y., Pozzo, A.D., Potop-Butucaru, M., Piergiovanni, S.T.: Correctness and fairness of tendermint-core blockchains. Technical Report (2018). arXiv:1805.08429
6. Amoussou-Guenou, Y., Pozzo, A.D., Potop-Butucaru, M., Tucci-Piergiovanni, S.: Dissecting tendermint. In: Proceedings of the 7th Edition of The International Conference on Networked Systems (2019)

7. Armknecht, F., Karame, G.O., Mandal, A., Youssef, F., Zenner, E.: Ripple: overview and outlook. In: International Conference on Trust and Trustworthy Computing, pp. 163–180. Springer (2015)
8. Aublin, P.-L., Guerraoui, R., Knežević, N., Quéma, V., Vukolić M.: The next 700 BFT protocols. ACM Trans. Comput. Syst. **32**(4), 12:1–12:45 (2015). Jan
9. Balasubramanian, A.R., Esparza, J., Lazic, M.: Complexity of verification and synthesis of threshold automata. In: ATVA, pp. 144–160 (2020)
10. Berman P., Garay, J.A.: Asymptotically optimal distributed consensus (extended abstract). In: ICALP, pp. 80–94 (1989)
11. Bertrand, N., Gramoli, V., Konnov, I., Lazic, M., Tholoniat, P., Widder, J.: Compositional verification of byzantine consensus. Technical Report hal-03158911v1 (2021). HAL
12. Bertrand, N., Konnov, I., Lazic, M., Widder, J.: Verification of randomized consensus algorithms under round-rigid adversaries. In: CONCUR, pp. 33:1–33:15 (2019)
13. Bertrand, N., Konnov, I., Lazic, M., Widder, J.: Verification of randomized distributed algorithms under round-rigid adversaries. In: CONCUR (2019)
14. Biely, M., Schmid, U., Weiss, B.: Synchronous consensus under hybrid process and link failures. Theor. Comput. Sci. **412**(40), 5602–5630 (2011). Sept
15. Bracha, G., Toueg, S.: Asynchronous consensus and broadcast protocols. J. ACM **32**(4), 824–840 (1985). Oct
16. Brown, B.: xRapid: everything you need to know about ripple's crypto service (now live) (Jan 2019). https://blockexplorer.com/news/what-is-xrapid/
17. Buchman, E., Kwon, J., Milosevic, Z.: The latest gossip on BFT consensus. Technical report, Tendermint (2018)
18. Buterin, V., Griffith, V.: Casper the friendly finality gadget. Technical Report (Jan 2019). arXiv:1710.09437v4
19. Cachin, C., Kursawe, K., Shoup, V.: Random oracles in constantipole: practical asynchronous byzantine agreement using cryptography (extended abstract). In: PODC, pp. 123–132 (2000)
20. Cachin, C., Vukolić, M.: Blockchains consensus protocols in the wild (2017). arXiv:1707.01873
21. Cachin, C., Zanolini, L.: Asymmetric byzantine consensus. Technical Report (2020). arXiv:2005.08795
22. Castro, M., Liskov, B.: Practical byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. **20**(4), 398–461 (2002). Nov
23. Charron-Bost, B., Debrat, H., Merz, S.: Formal verification of consensus algorithms tolerating malicious faults. In: Stabilization, Safety, and Security of Distributed Systems-13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings, pp. 120–134 (2011)
24. Chase, B., MacBrough, E.: Analysis of the xrp ledger consensus protocol. Technical Report (2018). arXiv:1802.07242v1. (Feb. 2018)
25. Chase, J.M.: Quorum whitepaper (Aug 2018). https://github.com/jpmorganchase/quorum/blob/master/docs/Quorum%20Whitepaper%20v0.2.pdf
26. Civit, P., Gilbert, S., Gramoli, V.: Brief announcement: Polygraph: accountable byzantine agreement. In: DISC (2020)
27. Civit, P., Gilbert, S., Gramoli, V.: Polygraph: accountable byzantine agreement. In: ICDCS (Jul. 2021)
28. Civit, P., Gramoli, V., Gilbert, S: Polygraph: accountable byzantine agreement. Technical Report 2019/587, ePrint (2019). https://eprint.iacr.org/2019/587.pdf
29. Cousineau, D., Doligez, D., Lamport, L., Merz, S., Ricketts, D., Vanzetto, H.: TLA + proofs. In: FM, pp. 147–154 (2012)
30. Crain, T., Gramoli, V., Larrea, M., Raynal, M.: DBFT: efficient leaderless Byzantine consensus and its applications to blockchains. In NCA, IEEE (2018)
31. Crain, T., Natoli, C., Gramoli, V.: Evaluating the Red Belly blockchain. Technical Report (2018). arXiv:1812.11747
32. Crain, T., Natoli, C., Gramoli, V.: Red belly: a secure, fair and scalable open blockchain. In: Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P'21), pp. 1501–1518 (May 2021)

33. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Monographs in Computer Science. Springer (1999)
34. Dragoi, C., Henzinger, T.A., Zufferey, D.: PSync: a partially synchronous language for fault-tolerant distributed algorithms. In: POPL, pp. 400–415 (2016)
35. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (1988). Apr
36. Ekparinya, P., Gramoli, V., Jourjon, G.: The attack of the clones against proof-of authority. In: Community Ethereum Development Conference (EDCON'19) (2019). (Apr. 2019, Presentation)
37. Ekparinya, P., Gramoli, V., Jourjon G.: The Attack of the clones against proof-of-authority. In: Proceedings of the Network and Distributed Systems Security Symposium (NDSS'20). Internet Society (Feb. 2020)
38. Ethereum: Ethereum 2.0 (serenity) phases (2019). https://docs.ethhub.io/ethereum-roadmap/ethereum-2.0/eth-2.0-phases/. (23 Aug. 2019)
39. Golan-Gueta, G., Abraham, I., Grossman, S., Malkhi, D., Pinkas, B., Reiter, M.K., Seredinschi, D., Tamir, O., Tomescu, A.: SBFT: a scalable decentralized trust infrastructure for blockchains. Technical Report (2018). arXiv:1804.01626
40. Gramoli, V.: On the danger of private blockchains. In: Workshop on Distributed Cryptocurrencies and Consensus Ledgers (2016)
41. Guerraoui, R., Kuznetsov, P., Monti, M., Pavlovič, M., Seredinschi, D.-A.: The consensus number of a cryptocurrency. In: PODC, pp. 307–316 (2019)
42. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: Ironfleet: proving practical distributed systems correct. In: SOSP, pp. 1–17 (2015)
43. Igor Barinov, P.K.: Viktor Baranov. POA network white paper (Sept. 2018). https://github.com/poanetwork/wiki/wiki/POA-Network-Whitepaper
44. John, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In: FMCAD, pp. 201–209 (2013)
45. Konnov, I., Lazić, M., Veith, H., Widder, J.: A short counter example property for safety and liveness verification of fault-tolerant distributed algorithms. In: POPL, pp. 719–734 (2017)
46. Konnov, I., Veith, H., Widder, J.: SMT and POR beat counter abstraction: parameterized model checking of threshold-based distributed algorithms. In: CAV, vol. 9206. LNCS, pp. 85–102 (2015)
47. Konnov, I., Widder, J.: ByMC: byzantine model checker. In: ISoLA, pp. 327–342 (2018)
48. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: speculative byzantine fault tolerance. ACM Trans. Comput. Syst. **27**(4), 7:1–7:39 (2010). Jan
49. Kwon, J.: Tendermint: consensus without mining-draft v.0.6 (2014)
50. Lamport, L.: Byzantizing paxos by refinement. In: DISC, pp. 211–224 (2011)
51. Lazic, M., Konnov, I., Widder, J., Bloem, R.: Synthesis of distributed algorithms with parameterized threshold guards. In: OPODIS, pp. 32:1–32:20 (2017)
52. Lin, Y.-T.: Istanbul byzantine fault tolerance-eip 650 (2019). https://github.com/ethereum/EIPs/issues/650. (21 Aug. 2019)
53. Losa, G., Dodds, M.: On the formal verification of the stellar consensus protocol. In: 2nd Workshop on Formal Methods for Blockchains, FMBC@CAV 2020, pp. 9:1–9:9 (2020)
54. Lynch, N.: Input/output automata: basic, timed, hybrid, probabilistic, dynamic,... In: Amadio R.L.D. (ed.) Proceedings of the Conference on Concurrency Theory (CONCUR), vol. 2761. Lecture Notes in Computer Science (2003)
55. Maric, O., Sprenger, C., Basin, D.A.: Cutoff bounds for consensus algorithms. In: Proceedings fo the Computer Aided Verification Conference, CAV, pp. 217–237 (2017)
56. Miller, A., Xia, Y., Croman, K., Shi, E., Song, D.: The honey badger of BFT protocols. In: CCS (2016)
57. Mostéfaoui, A., Moumen, H., Raynal, M.: Signature-free asynchronous Byzantine consensus with $T < N/3$ and $O(N^2)$ messages. In: PODC, pp. 2–9 (2014)
58. Mostéfaoui, A., Moumen, H., Raynal, M.: Signature-free asynchronous binary Byzantine consensus with $t < n/3$, $O(n^2)$ messages and $O(1)$ expected time. J. ACM (2015)

59. Mostéfaoui, A., Mourgaya, E., Parvédy, P.R., Raynal, M.: Evaluating the condition-based approach to solve consensus. In: DSN, pp. 541–550 (2003)
60. Mostéfaoui, A., Rajsbaum, S., Raynal, M.: Conditions on input vectors for consensus solvability in asynchronous distributed systems. J. ACM **50**(6), 922–954 (2003). Nov
61. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008)
62. Neu, J., Tas, E.N., Tse, D.: Ebb-and-flow protocols: a resolution of the availability-finality dilemma. In: Proceedings of the 42nd IEEE Symposium on Security and Privacy (S& P'21) (2021). May 2021
63. Newcombe, C.: Why amazon chose TLA+. In: ABZ, pp. 25–39 (2014)
64. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL-A Proof Assistant for Higher-Order Logic, vol. 2283. Lecture Notes in Computer Science. Springer (2002)
65. Pease, M.C., Shostak, R.E., Lamport, L.: Reaching agreement in the presence of faults. J. ACM **27**(2), 228–234 (1980)
66. Rahli, V., Guaspari, D., Bickford, M., Constable, R.L.: Formal specification, verification, and implementation of fault-tolerant systems using EventML. *ECEASST*, 72, 2015
67. Ranchal-Pedrosa, A., Gramoli, V.: Blockchain is dead, long live blockchain! accountable state machine replication for longlasting blockchain. Technical Report (2020). arXiv:abs/2007.10541
68. Saltini, R.: Correctness analysis of IBFT. Technical Report (Jan. 2019). arXiv:1901.07160v1
69. Schwartz, D., Youngs, N., Britto, A.: The ripple protocol consensus algorithm, vol. 5. Ripple Labs Inc., White Paper (2014)
70. Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and proving with distributed protocols. In: PACMPL, 2(POPL), 28:1–28:30 (2018)
71. Song, Y.J., van Renesse, R.: Bosco: one-step byzantine asynchronous consensus. In: DISC, pp. 438–450 (2008)
72. Stoilkovska, I., Konnov, I., Widder, J., Zuleger, F.: Verifying safety of synchronous fault-tolerant algorithms by bounded model checking. In: TACAS, pp. 357–374 (2019)
73. Sutra,P.: On the correctness of egalitarian Paxos. Inf. Proc. Lett. **156** (2020)
74. Thomas, S., Schwartz, E.: A protocol for interledger payments (2015). https://interledger.org/interledger.pdf
75. Tsuchiya, T., Schiper, A.: Using bounded model checking to verify consensus algorithms. In: Taubenfeld, G. (ed.) Distributed Computing, pp. 466–480 (2008)
76. Tsuchiya, T., Schiper, A.: Verification of consensus algorithms using satisfiability solving. Distributed Comput. **23**(5–6), 341–358 (2011)
77. von Gleissenthall, K., Kici, R.G., Bakst, A., Stefan, D., Jhala, R.: Pretend synchrony: synchronous verification of asynchronous distributed programs. In: PACMPL, vol. 3(POPL), pp. 59:1–59:30 (2019)
78. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.E.: Verdi: a framework for implementing and formally verifying distributed systems. In: PLDI, pp. 357–368 (2015)
79. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA$^+$ specifications. In: CHARME, pp. 54–66 (1999)
80. Zamfir, V., Rush, N., Asgaonkar, A., Piliouras, G.: Introducing the "minimal" cbc casper family of consensus protocols (2018). https://github.com/cbc-casper/cbc-casper-paper/blob/master/cbc-casper-paper-draft.pdf. (21 Aug. 2019)