SURVEY ON PARAMETERIZED VERIFICATION WITH THRESHOLD AUTOMATA AND THE BYZANTINE MODEL CHECKER

IGOR KONNOV [1], MARIJANA LAZIĆ [2], ILINA STOILKOVSKA [3], AND JOSEF WIDDER [4]

Informal Systems, Vienna, Austria [1]

TU Munich, Munich, Germany [2]

TU Wien & Informal Systems, Vienna, Austria [3]

Informal Systems, Vienna, Austria [4]

Abstract. Threshold guards are a basic primitive of many fault-tolerant algorithms that solve classical problems of distributed computing, such as reliable broadcast, two-phase commit, and consensus. Moreover, threshold guards can be found in recent blockchain algorithms such as Tendermint consensus. In this article, we give an overview of the techniques implemented in Byzantine Model Checker (ByMC). ByMC implements several techniques for automatic verification of threshold-guarded distributed algorithms. These algorithms have the following features: (1) up to t of processes may crash or behave Byzantine; (2) the correct processes count messages and make progress when they receive sufficiently many messages, e.g., at least $t+1$; (3) the number n of processes in the system is a parameter, as well as t; (4) and the parameters are restricted by a resilience condition, e.g., $n > 3t$. Traditionally, these algorithms were implemented in distributed systems with up to ten participating processes. Nowadays, they are implemented in distributed systems that involve hundreds or thousands of processes. To make sure that these algorithms are still correct for that scale, it is imperative to verify them for all possible values of the parameters.

1. Introduction

The recent advent of blockchain technologies [72, 34, 2, 21, 91, 24] has brought fault-tolerant distributed algorithms to the spotlight of computer science and software engineering. In particular, due to the huge amount of funds managed by blockchains, it is crucial that their software is free of bugs. At the same time, these systems are characterized by a large number of participants. Thus, automated verification methods face the well-known state space explosion problem. Furthermore, the well-known undecidability results for the

Supported by Interchain Foundation (Switzerland) [1].

Supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreement No 787367 (PaVeS) [2].

Supported by Interchain Foundation (Switzerland) and by the Austrian Science Fund (FWF) via the Doctoral College LogiCS W1255 [3].

Supported by Interchain Foundation (Switzerland) [4].

Preprint submitted to
Logical Methods in Computer Science

verification of parameterized systems [4, 88, 41, 42, 16] apply in this setting. One way to circumvent these problems is to develop domain specific methods that work for a specific subclass of systems.

In this article, we survey verification techniques for fault-tolerant distributed algorithms. As an example, consider a blockchain system, where a blockchain algorithm ensures coordination of the processes participating in the system. We observe that to do so, the processes need to solve a coordination problem called *atomic (or, total order) broadcast* [47], that is, every process delivers the same transactions in the same order. To achieve that, we typically need a *resilience condition* that restricts the fraction of processes that may be faulty [74]. The techniques we survey deal with the concepts of broadcast and atomic broadcast under resilience conditions.

While Bitcoin [72] was a new approach to consensus, several Blockchain systems like Tendermint [21] and HotStuff [91] are modern implementations that are built on these classic Byzantine fault tolerance concepts. While the techniques we describe here address in part the challenges for the verification of such systems. We discuss open challenges in Section 8.

In addition to practical importance, the reasons for the long-standing interest [65, 62, 74, 43] in distributed systems is that distributed consensus is non-trivial in two aspects:

(1) Most coordination problems are impossible to solve without imposing constraints on the environment, e.g., an upper bound on the fraction of faulty processes, assumptions on the behavior of faulty processes, or bounds on message delays and processing speeds (i.e., restricting interleavings) [74, 43, 37].
(2) Designing correct solutions is hard, owing to the huge state and execution space, and the complex interplay of assumptions mentioned in Point 1. Thus, even published protocols may contain bugs, as reported, e.g., by [66, 68].

Due to the impossibility of asynchronous fault-tolerant consensus [43], much of the research focuses one what kinds of problems are solvable in asynchronous systems (e.g., some forms of reliable broadcast) or what kinds of systems allow to solve consensus. In Section 2 we will survey some of the most fundamental system assumptions that allow to solve problems in the presence of faults and example algorithms. In Sections 3 to 5 we will discuss how these algorithms can be formalized in threshold automata and how they can be automatically verified. Indeed threshold automata represent an abstraction of distributed algorithms. In Section 6 we discuss how this abstraction can be automatically generated from a formalization close to the algorithm descriptions in the literature. We then present in Section 7 how out tool ByMC evolved in the last years and which techniques were implemented. While our standard benchmarks were classic fault-tolerant distributed algorithms from the literature, we demonstrate in Section 8 how ByMC can be used to analyze Tendermint, a state-of-the art consensus algorithms used in the Cosmos blockchain ecosystem.

2. Threshold-guarded distributed algorithms

In a classic survey, Schneider [79] explains replicated state machines by the following notion of replica coordination that consists of two properties:

Agreement.: "Every non-faulty state machine replica receives every request."

Order.: "Every non-faulty state machine replica processes the requests it receives in the same relative order."

```
1  int v:=input($\{0,1\}$);
2  bool accept:=false;
3  while (true) do {
4    if (v = 1) then send <ECHO> to all;
5    receive messages from other processes;
6    if received <ECHO> from $\geq$ t + 1 processes
7      then v:=1;
8    if received <ECHO> from $\geq$ n − t processes
9      then accept:=true;
10 }
```

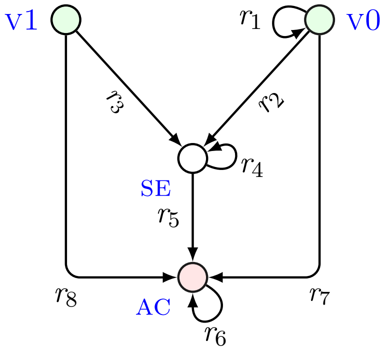

Figure 1: Pseudo code of reliable broadcast à la [83] and its threshold automaton.

In Schneider's approach [79], the specification of *Agreement* can be solved using an algorithm for reliable broadcast [47]. The processes can use a consensus algorithm [39, 29, 26] to establish the *Order* property. For instance, the atomic broadcast algorithm from [26] contains these two sub-algorithms.

The simplest canonical system model that allows one to solve consensus is the synchronous one, and we discuss it in Section 2.1. A second elegant way to circumvent the impossibility of [43] is by replacing liveness with almost sure termination, that is, a probabilistic guarantee. We review this approach in Section 2.3. In fact, reliable broadcast can be solved with an asynchronous distributed algorithm. We discuss their characteristics in Section 2.2.

2.1. **Synchronous algorithms.** A classic example of a fault-tolerant distributed algorithm is the broadcasting algorithm by Srikanth & Toueg [82]. The description of its code is given in Figure 1. As is typical for distributed algorithms, the semantics is not visible from the pseudo code. In fact, we use the same pseudo-code to describe its asynchronous variant later in Section 2.2.

The algorithm satisfies the Agreement property mentioned above. In a distributed system comprising reliable servers, which do not fail and do not lose messages, this property is easy to achieve. If a server receives a request, it sends the request to all other servers. As messages are delivered reliably, every request will eventually be received by every server. The problems comes with faults. Srikanth and Toueg studied Byzantine failures, where faulty servers may send messages only to a subset of the servers (or even send conflicting data). In this scenario, two servers may receive different requests. The algorithm in Figure 1 addresses this problem, by forwarding message content received from other servers and only accepting a message content when it was received from a quorum of servers. For each message content m, one instance of this algorithm is executed. Initially the variable v captures whether a process has received m, and it stores the value 1 if this is the case. A process that has received m sends ECHO to all (line 4). In an implementation, the message would be of the form (ECHO, m), that is, it would be tagged with ECHO, and carry the content m to distinguish different instances running in parallel; also it would suffice to send the message once instead of sending it in each iteration. If the guard in line 6 evaluates to true at a server p, then p has received $t+1$ ECHO messages, which means that at least one correct process has forwarded the message. This triggers the server p to also forward the

```
best := input_value;
for each round 1 through $\lfloor t/k \rfloor + 1$ do {
   broadcast best;
   receive values $b_1, \dots b_\ell$ from others;
   best := min $\{b_1, \dots b_\ell\}$;
}
choose best;
```

Figure 2: Pseudo code of *FloodMin* from [30]

message. If a server p receives $n - t$ ECHO messages, it finally accepts the request stored in m due to line 8. The reason this algorithm works is that the combination of $n - t$, $t + 1$, and $n > 3t$ ensures that if one correct processes has $n - t$ ECHO messages, every other correct process will eventually receive at least $t + 1$ (there are $t + 1$ correct processes among any $n - t$ processes). This implies that every correct process will forward the message, and since there are at least $n - t$ correct processes, every correct will accept. However, this arithmetics over parameters is subtle and error-prone. To overcome this, our verification techniques focus on threshold expressions and resilience conditions.

In the above discussion, we were imprecise about the code semantics. In this section we consider the synchronous semantics: All correct processes execute the code line-by-line in lock-step. One loop iteration is called one *round*. A message sent by a correct process to a correct process is received within the same round. Then after sending and receiving messages in lock-step, all correct processes continue by evaluating the guards, before they all proceed to the next round. Because this semantics ensures that all processes move together, and all messages are received within the next rounds, no additional fairness constraints are needed to ensure liveness (something good eventually happens). In practice, this approach is often considered slow and expensive, as it has to be implemented with timeouts that are aligned to worst case message delays (which can be very slow in real networks). However, synchronous semantics offers a high-level abstraction that allows one to design algorithms easier.

Figure 2 shows an example of another synchronous algorithm. This algorithm is run by n replicated processes, up to t of which may fail by crashing, that is, by prematurely halting. It solves the k-set agreement problem, that is, out of the n initial values each process decides on one value, such that the number of different decision values is at most k. By setting $k = 1$, we obtain that there can be exactly one decision value, which coincides with the definition of consensus. In contrast to the reliable broadcast above, it runs for a finite number of rounds. The number of loop iterations $\lfloor t/k \rfloor + 1$ of the FloodMin algorithm has been designed such that it ensures that there is at least one clean round in which at most $k - 1$ processes crash. When we consider consensus, this means there is a round in which no process crashes, such that all processes receive the same values $b_1, \dots b_\ell$. As a result, during that round all processes set *best* to the same value.

2.2. **Asynchronous algorithms.** We now discuss the asynchronous semantics of the code in Figure 1: at each time point, exactly one processes performs a step. That is, the steps of the processes are interleaved. In the example, one may interpret this as one code line being an atomic unit of execution at a process. In the “receive” statement, a process takes some

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

Figure 3: Pseudo code of Ben-Or's algorithm for Byzantine faults

messages out of the incoming message buffer: possibly no message, and not necessarily all messages that are in the buffer. The "send to all" then places one message in the message buffers of all the other processes. Often, the asynchronous semantics is considered more coarse-grained, e.g., a step consists of receiving messages, updating the state, and sending one or more messages.

As we do not restrict which messages are taken out of the buffer during a step, we cannot bound the time needed for message transmission. Moreover, we do not restrict the order, in which processes have to take steps, so we cannot bound the time between two steps of a single process. Typically, we are interested in verifying safety (nothing bad ever happens) under these conditions.

However, for liveness this is problematic. We need messages to be delivered eventually, and correct processes to take steps from time to time. That is, liveness is typically preconditioned by fairness guarantees: every correct processes takes infinitely many steps and every message sent from a correct process to a correct process is eventually received. These constraints are sufficient for broadcast, while for consensus they are not.

2.3. **Randomized algorithms.** A prominent example is Ben-Or's fault-tolerant binary consensus [8] algorithm in Figure 3. It circumvents the impossibility of asynchronous consensus [43] by relaxing the termination requirement to almost-sure termination, i.e., termination with probability 1. Here, the processes execute an infinite sequence of asynchronous rounds. While the algorithm is executed under asynchronous semantics, the processes have a local variable r that stores the round number, and use it to tag the messages that they send round r. Observe that the algorithm only operates on messages from the current round (the guards only count messages tagged with r). Asynchronous algorithms with this feature are called *communication closed* [40, 32]. In the case of Ben-Or's algorithm, each round consists of two stages where the processes first exchange messages tagged with R, wait until the number of received messages reaches a certain threshold (the expression over parameters in line 5) and then exchange messages tagged with P. As in the previous examples, n is the number of processes, among which at most t may crash or be Byzantine. The thresholds $n - t$, $(n + t)/2$ and $t + 1$ in combination with the resilience condition $n > 5t$ ensure that no two correct processes ever decide on different values. If there is no "strong majority" for a value in line 13, a process chooses a new value by tossing a coin in line 16.

3. Parameterized Verification of Synchronous Algorithms

In [84], we introduced the synchronous variant of threshold automata, and studied their applicability and limitations for verification of synchronous fault-tolerant distributed algorithms. We showed that the parameterized reachability problem for synchronous threshold automata is undecidable. Nevertheless, we observed that counter systems of many synchronous fault-tolerant distributed algorithms have bounded diameters, even though the algorithms are parameterized by the number of processes. Hence, bounded model checking can be used for verifying these algorithms. We briefly discuss these results in the following.

3.1. **Synchronous Threshold Automata.** In a synchronous algorithm, the processes execute the send, receive, and local computation steps in lock-step. Consider the synchronous reliable broadcast algorithm from [83], whose pseudocode is given in Figure 1 (left). A *synchronous threshold automaton (STA)* that encodes the pseudocode of this algorithm is given in Figure 1 (right). The STA models the loop body of the pseudo code: one iteration of the loop is expressed as an STA edge that connects the locations before and after a loop iteration.

The semantics of the synchronous threshold automaton is defined in terms of a counter system. For each location $\ell_i \in \{\mathrm{V0}, \mathrm{V1}, \mathrm{SE}, \mathrm{AC}\}$ (a node in the graph), we have a counter κ_i that stores the number of processes located in ℓ_i. The counter system is parameterized in two ways: (i) in the number of processes n, the number of faults f, and the upper bound on the number of faults t, (ii) the expressions in the guards contain n, t, and f. Every system transition moves all processes simultaneously; potentially using a different rule for each process (depicted by an edge in the figure), provided that the rule guards evaluate to true. The guards compare a sum of counters to a linear combination of parameters. Processes send messages based on their current locations. Hence, we use the number of processes in given locations to test how many messages of a certain type have been sent in the previous round. However, the pseudo code in Figure 1 is predicated by received messages rather than by sent messages. This algorithm is designed to tolerate Byzantine-faulty processes, which may send corrupt messages to some correct processes. Thus, the number of received messages may deviate from the number of correct processes that sent a message. For example, if the guard in line 6 evaluates to true, the $t + 1$ received messages may contain up to f messages from the faulty processes. If i correct processes send ECHO, for $1 \le i \le t$, the faulty processes may "help" some correct processes to pass over the $t + 1$ threshold. That is, the effect of the f faulty processes on the correct processes is captured by the ”$-f$” component in the guards. As a result, we run only the correct processes, so that a system consists of $n - f$ copies of the STA.

For example, in the STA in Figure 1, processes send a message if they are in a location V1, SE, or AC. Thus, the guards compare the number of processes in a location V1, SE, or AC, which we denote by $\#\{\mathrm{V1}, \mathrm{SE}, \mathrm{AC}\}$, to some linear expression over the parameters, called a threshold. The assignment v:=1 in line 7 is modeled by the rule r_2, guarded with $\phi_1 \equiv \#\{\mathrm{V1}, \mathrm{SE}, \mathrm{AC}\} \ge t + 1 - f$. This guard evaluates to true if he number of processes in location V1, SE, or AC is greater than or equal to $t + 1 - f$. The implicit "else" branch between lines 6 and 8 is modeled by the rule r_1, guarded with $\phi_3 \equiv \#\{\mathrm{V1}, \mathrm{SE}, \mathrm{AC}\} < t + 1$. The effect of the faulty processes is captured by both the rules r_1 and r_2 being enabled. Similarly, the rules r_5, r_7, r_8 are guarded with the guard $\phi_2 \equiv \#\{\mathrm{V1}, \mathrm{SE}, \mathrm{AC}\} \ge n - t - f$, which is true when the number of process in one of V1, SE, or AC is greater or equal to

Table 1: A long execution of reliable broadcast and the short representative.

Process	σ_0	σ_1	σ_2	…	σ_{t+1}	σ_{t+2}	σ_{t+3}
1	V1	SE	SE	…	SE	SE	AC
2	V0	V0	SE	…	SE	SE	AC
…	…						
$t+1$	V0	V0	V0	…	SE	SE	AC
…	…						
$n-f$	V0	V0	V0	…	V0	SE	AC

Process	σ'_0	σ'_1	σ'_2
1	V1	SE	AC
2	V0	SE	AC
…	…		
$t+1$	V0	SE	AC
…	…		
$n-f$	V0	SE	AC

$n - t - f$, while the rules r_3, r_4 are guarded with $\phi_4 \equiv \#\{\text{V}1, \text{SE}, \text{AC}\} < n - t$. The rule r_6 is unguarded, i.e., its guard is $\top$.

3.2. **Bounded Diameter.** An example execution of the synchronous reliable broadcast algorithm is depicted in Table 1 on the left. Observe that the guards of the rules r_1 and r_2 are both enabled in the configuration σ_0 of the counter system for the STA in Figure 1. One STA uses r_2 to go to SE while the others use the self-loop r_1 to stay in V0. As both rules remain enabled, in every round one copy of STA can go to SE. Hence, the configuration σ_{t+1} has $t+1$ correct STA in location SE and the rule r_1 becomes disabled. Then, all remaining STA go to SE and then finally to AC. This execution depends on the parameter t, which implies that the length of this execution grows with t and is thus unbounded. (We note that we can obtain longer executions, if some STA use the rule r_4). On the right, we see an execution where all copies of STA immediately move to SE via rule r_2. That is, while the configuration σ_{t+3} is reached by a long execution on the left, it is reached in just two steps on the right (observe that $\sigma'_2 = \sigma_{t+3}$). We are interested in whether there is a natural number k (independent of n, t and f) such that we can always shorten executions to executions of length $\leq k$. (By length, we mean the number of transitions in an execution.) In such a case, we say that the STA has *bounded diameter*. We adapt the definition of diameter from [15], and introduce an SMT-based procedure for computing the diameter of the counter system. The procedure enumerates candidates for the diameter bound, and checks (by calling an SMT solver) if the number is indeed the diameter; if it finds such a bound, it terminates.

3.3. **Bounded Model Checking.** The existence of a bounded diameter motivates the use of bounded model checking, as safety verification can be reduced to checking the violation of a safety property in executions with length up to the diameter. Crucially, this approach is complete: if an execution reaches a bad configuration, this bad configuration is already reached by an execution of bounded length. Thus, once the diameter is found, we encode the violation of a safety property using a formula in Presburger arithmetic, and use an SMT to check for violations.

The SMT queries that are used for computing the diameter and encoding the violation of the safety properties contain quantifiers for dealing with the parameters symbolically. Surprisingly, performance of the SMT solvers on these queries is very good, reflecting the recent progress in dealing with quantified queries. We found that the diameter bounds of synchronous algorithms in the literature are tiny (from 1 to 8), which makes our approach applicable in practice. The verified algorithms are given in Section 7.

3.4. **Undecidability.** In [84], we showed that the parameterized reachability problem is in general undecidable for STA. In particular, this implies that some STA have unbounded diameters. We identified a class of STA which in theory have bounded diameters. For some STA outside of this class, our SMT-based procedure still can automatically find the diameter. Remarkably, the SMT-based procedure gives us the diameters that are independent of the parameters.

4. Parameterized Verification of Asynchronous Algorithms

4.1. **Asynchronous Threshold Automata.** Similarly as in STA, nodes in asynchronous threshold automata (TA) represent locations of processes, and edges represent local transitions. What makes a difference between an STA and a TA are shared variables and labels on edges that have a form $\gamma \mapsto \mathsf{act}$. A process moves along an edge labelled by $\gamma \mapsto \mathsf{act}$ and performs an action act, only if the condition γ, called a *threshold guard*, evaluates to true.

We model reliable broadcast [82] using the same threshold automaton from Figure 1 but with different edge labels in comparison to the STA. We use a shared variable x to capture the number of ECHO messages sent by correct processes. We have two threshold guards: $\gamma_1\colon x \geq (t+1) - f$ and $\gamma_2\colon x \geq (n-t) - f$. Depending on the initial value of a correct process, 0 or 1, the process is initially either in location $\textsc{v}0$ or in $\textsc{v}1$. If its value is 1 a process broadcasts ECHO, and executes the rule $r_3\colon \textsc{true} \mapsto x{+}{+}$. This is modelled by a process moving from $\textsc{v}1$ to $\textsc{se}$ and increasing the value of x. If its value is 0, it has to wait to receive enough messages, i.e., it waits for γ_1 to become true, and then it broadcasts the ECHO message and moves to location $\textsc{se}$. Thus, r_2 is labelled by $\gamma_1 \mapsto x{+}{+}$. Finally, once a process has γ_2-enough ECHO messages, it sets accept to true and moves to $\textsc{ac}$. Thus, r_5 is labelled by γ_2, whereas r_7 and r_8 by $\gamma_2 \mapsto x{+}{+}$.

4.2. **Counter Systems.** Similarly to STA, the semantics of TA is captured by counter systems. Instead of storing the location of each process, we count the number of processes in each location, as all processes are identical. We also store the values of the shared variables, which are incremented as the processes execute the rules. Therefore, a configuration comprises (i) values of the counters for each location, (ii) values of the shared variables, and (iii) parameter values. A configuration is initial if all processes are in initial locations, here $\textsc{v}0$ or $\textsc{v}1$, and all shared variables have value 0 (here $x = 0$). A transition of a process along an edge from location ℓ to location ℓ' — labelled by $\gamma \mapsto \mathsf{act}$ — is modelled by the configuration update as follows: (i) the counter of ℓ is decreased by 1, and the counter of ℓ' is increased by 1, (ii) shared variables are updated according to the action act, and (iii) parameter values are unchanged. The key ingredient of our technique is acceleration of transitions, that is, many processes may move along the same edge simultaneously. In the resulting configuration, the counters and shared variables are updated depending on the number of processes that participate in the transition. It is important to notice that any accelerated transition can be encoded in SMT.

4.3. **Reachability.** In [53], we determine a finite set of execution "patterns", and then analyse each pattern separately. These patterns restrict the order in which the threshold guards become true (if ever). Namely, we observe how the set of guards that evaluate to true changes along each execution. In our example TA for the reliable broadcast algorithm, given in Figure 1, there are two (non-trivial) guards, γ_1 and γ_2. Initially, both evaluate to false, as $x = 0$. During an execution, none, one, or both of them become true. Note that once they become true, they can never evaluate to false again, as the number of sent messages x cannot decrease. Thus, there is a finite set of execution patterns.

For instance, a pattern $\{\} \dots \{\gamma_1\} \dots \{\gamma_1, \gamma_2\}$ captures all finite executions τ that can be represented as $\tau = \tau_1 \cdot t_1 \cdot \tau_2 \cdot t_2 \cdot \tau_3$, where τ_1, τ_2, τ_3 are sub-executions of τ, and t_1 and t_2 are transitions. No threshold guard is enabled in a configuration visited by τ_1, and only γ_1 is enabled in all configurations visited by τ_2. Both guards are enabled in all configurations visited by τ_3, and t_1 and t_2 change the evaluation of the guards. Another pattern $\{\} \dots \{\gamma_2\} \dots \{\gamma_1, \gamma_2\}$ enables γ_2 before γ_1. The third pattern $\{\} \dots \{\gamma_1\}$ never enables γ_2.

To perform verification, we have to analyse all execution patterns. For each pattern, we construct a so-called *schema* defined as a sequence of accelerated transitions, whose free variables are the number of processes that execute the transitions and the parameter values. In Figure 1, the transitions are modelled by edges denoted with $r_i,\ i \in \{1, \dots, 8\}$. For instance, the pattern $\{\} \dots \{\gamma_1\}$ produces the schema:

$$\mathcal{S} = \{\}\ \underbrace{r_1, r_3}_{\tau_1}, \underbrace{r_3}_{t_1}\ \{\gamma_1\}\ \underbrace{r_1, r_2, r_3, r_4}_{\tau_2}\ \{\gamma_1\}\ .$$

There are two segments, τ_1 and τ_2, corresponding to $\{\}$ and $\{\gamma_1\}$, respectively. In each of them we list all the rules that can be executed according to the guards that evaluate to true, in a fixed natural order: only r_1 and r_3 can be executed if no guard is enabled, and r_1, r_2, r_3, r_4 if only the guard γ_1 holds true. Additionally, we have to list all the candidate rules for t_1 that can change the evaluation of the guards. In our example, only r_3 can enable the guard γ_1.

We say that an execution follows the schema $\mathcal{S}$ if its transitions appear in the same order as in $\mathcal{S}$, but they are accelerated (every transition is executed by a number of processes, possibly zero). For example, if (r, k) denotes that k processes execute the rule r simultaneously, then the execution $\rho = (r_1, 2)(r_3, 3)(r_2, 2)(r_4, 1)$ follows $\mathcal{S}$, where the transitions of the form $(r, 0)$ are omitted. In this case, we prove that for each execution τ of pattern $\{\} \dots \{\gamma_1\}$, there is an execution τ' that follows the schema $\mathcal{S}$, and τ and τ' reach the same configuration (when executed from the same initial configuration). This is achieved by *mover analysis*: inside any segment in which the set of enabled guards is fixed, we can swap adjacent transitions (that are not in a natural order). In this way, we gather all transitions of the same rule next to each other, and transform them into a single accelerated transition. For example, $\tau = (r_3, 2)(r_1, 1)(r_3, 1)(r_1, 1)(r_2, 1)(r_4, 1)(r_2, 1)$ can be transformed into $\tau' = \rho$ from above, and they reach the same configurations. Therefore, instead of checking reachability for all executions of the pattern $\{\} \dots \{\gamma_1\}$, it is sufficient to analyse reachability only for the executions that follow the schema $\mathcal{S}$.

Every schema is encoded as an SMT query over linear integer arithmetic with free variables for acceleration factors, parameters, and counters. An SMT model gives us an execution of the counter system, which typically disproves safety.

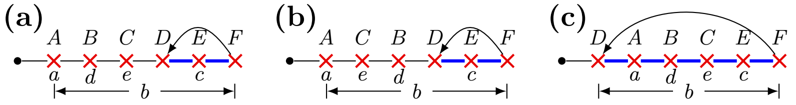

Figure 4: Three out of 18 shapes of lassos that satisfy the formula $\mathbf{F}\,(a \wedge \mathbf{F}\,d \wedge \mathbf{F}\,e \wedge \mathbf{G}\,b \wedge \mathbf{G}\,\mathbf{F}\,c)$. The crosses show cut points for: (A) formula $\mathbf{F}\,(a \wedge \mathbf{F}\,d \wedge \mathbf{F}\,e \wedge \mathbf{G}\,b \wedge \mathbf{G}\,\mathbf{F}\,c)$, (B) formula $\mathbf{F}\,d$, (C) formula $\mathbf{F}\,e$, (D) loop start, (E) formula $\mathbf{F}\,c$, and (F) loop end.

For example, consider the following reachability problem: Can the system reach a configuration with at least one process in ℓ_3? For each SMT query, we add the constraint that the counter of ℓ_3 is non-zero in the final configuration. If the query is satisfiable, then there is an execution where at least one process reaches ℓ_3. Otherwise, there is no such execution following the particular schema, where a process reaches ℓ_3. That is why we have to check all schemas.

4.4. **Safety and Liveness.** In [54] we introduced a fragment of Linear Temporal Logic called $\mathsf{ELTL}_{\mathsf{FT}}$. Its atomic propositions test location counters for zero. Moreover, this fragment only uses only two temporal operators: $\mathbf{F}$ (eventually) and $\mathbf{G}$ (globally). Our goal is to check whether there exists a counterexample to a temporal property, and thus formulas in this fragment represent negations of safety and liveness properties.

Our technique for verification of safety and liveness properties uses the reachability method as its basis. As before, we want to construct schemas that we can translate to SMT queries and check their satisfiability. Note that violations of liveness properties are infinite executions of a lasso shape, that is, $\tau \cdot \rho^{\omega}$, where τ and ρ are finite executions. Hence, we have to enumerate the patterns of lassos. These shapes depend not only on the values of the thresholds, but also on the evaluations of atomic propositions that appear in temporal properties. We single out configurations in which atomic propositions evaluate to true, and call them *cut points*, as they "cut" an execution into finitely many segments (see Figure 4).

We combine these cut points with those "cuts" in which the threshold guards become enabled (as in the reachability analysis). All the possible orderings in which the evaluations of threshold guards and formulas become true, give us a finite set of lasso patterns.

We construct a schema for each shape by first defining schemas for each of the segments between two adjacent cut points. On one hand, for reachability, it is sufficient to execute all enabled rules of that segment exactly once in the natural order. Thus, each sub-execution τ_i can be transformed into τ_i' that follows the segment's schema, so that τ_i and τ_i' reach the same final configuration. On the other hand, the safety and liveness properties reason about atomic propositions inside executions. To this end, we introduced a *property specific mover analysis* that allows us to construct schemas by executing all enabled rules a fixed number of times in a specific order. The number of rule repetitions depends on a temporal property; it is typically two or three.

For each lasso pattern we encode its schema in SMT and check its satisfiability. As $\mathsf{ELTL}_{\mathsf{FT}}$ formulas are negations of specifications, an SMT model gives us a counterexample. If no schema is satisfiable, the temporal property holds true.

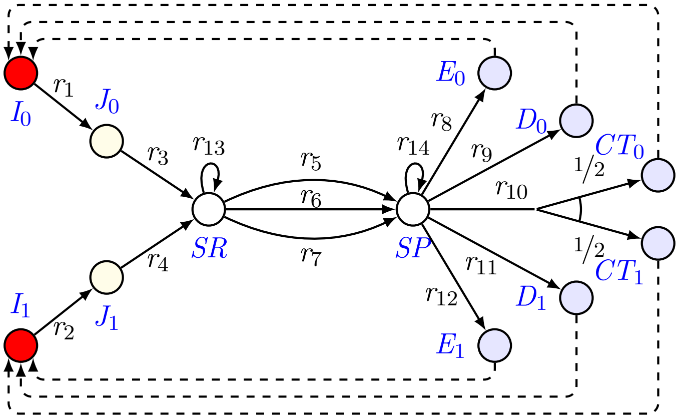

Figure 5: Ben-Or's alorithm as PTA with resilience condition $n > 3t \wedge t > 0 \wedge t \geq f \geq 0$.

Table 2: The rules of the PTA from Figure 5. We omit rules r_1, r_2, r_{13}, r_{14} as they have the trivial guard (*true*) and no update.

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

5. Parameterized Verification of Asynchronous Randomized Multi-Round Algorithms

5.1. **Probabilistic Threshold Automata.** Randomized algorithms typically have an unbounded number of asynchronous rounds and randomized choices. Probabilistic threshold automata (PTAs), introduced in [12], are extensions of asynchronous threshold automata that allow formalizing these features. A PTA modelling Ben-Or's algorithm from Figure 3 is shown in Figure 5. The behaviour of a process in a single round is modelled by the solid edges. Note that in this case threshold guards should be evaluated according to the values of shared variables, e.g., x_0 and x_1, in the observed round. The dashed edges model round switches: once a process reaches a final location in a round, it moves to an initial location of the next round. The coin toss is modelled by the branching rule r_{10}: a process in location SP can reach either location CT_0 or location CT_1 by moving along this fork, both with probability 1/2.

5.2. **Unboundedly many rounds.** In order to overcome the issue of unboundedly many rounds, we prove that we can verify PTAs by analysing a one-round automaton that fits in the framework of Section 4. In [12], we prove that one can reorder the transitions of any fair execution such that their round numbers are in a non-decreasing order. The obtained ordered execution is stutter-equivalent to the original one. Thus, both executions satisfy the same $\mathsf{LTL}_{\text{-}\mathsf{X}}$ properties over the atomic propositions of one round. In other words, the distributed system can be transformed to a sequential composition of one-round systems.

The main problem with isolating a one-round system is that the consensus specifications often talk about at least two different rounds. In this case we need to use round invariants that imply the specifications. For example, if we want to verify agreement, we have to check that no two processes decide different values, possibly in different rounds. We do this in two steps: (i) we check the round invariant that no process changes its decision from round to round, and (ii) we check that within a round no two processes disagree.

5.3. **Probabilistic properties.** The semantics of a probabilistic threshold automaton is an infinite-state Markov decision process (MDP), where the non-determinism is traditionally resolved by an adversary. In [12], we restrict our attention to so-called *round-rigid adversaries*, that is, fair adversaries that generate executions in which a process enters round $r+1$ only after all processes finished round r.

Verifying almost-sure termination under round-rigid adversaries calls for distinct arguments. Our methodology follows the lines of the manual proof of Ben-Or's consensus algorithm by Aguilera and Toueg [3]. However, our arguments are not specific to Ben-Or's algorithm, and apply to other randomized distributed algorithms (see Table 3). Compared to their paper-and-pencil proof, the threshold automata framework required us to provide a more formal setting and a more informative proof, also pinpointing the needed hypotheses. The crucial parts of our proof are automatically checked by the model checker ByMC.

5.4. **Weak adversaries.** The approach from Section 5.3 leaves a gap between round-rigid adversaries and the classed adversary definitions we find in distributed computing literature. This problem is addressed in [13] where the standard notion of a "weak adversary" is considered. Weak adversaries pose a formalization challenge in the counter system semantics of TAs. The reason is that these adversaries are defined over individual processes and messages; notions that do not exist in the counter system representation. As a result, a more concrete semantics of threshold automata was introduced, which explicitly captures processes, sets of received messages for each process, and threshold guards over the number of specific messages in these sets. For this semantics, [13] contains a reduction theorem from weak adversaries to round-rigid adversaries. While [13] does not contain a formalization of the abstraction step from the explicit model to (send) threshold automata, we conjecture that such a proof can be done based on the ideas that underly Section 6.1. Hence, verification results of ByMC can be lifted to algorithms scheduled by weak adversaries.

6. Modeling

6.1. **From Pseudocode to Threshold Automata.** Observe that the parameterized verification approaches, presented in Sections 3, 4, and 5, take as input threshold automata, whose guards are evaluated over the global state (the sent messages). When modeling threshold-guarded distributed algorithms with verification in mind, we are faced with a formalization gap between the threshold automata and the algorithm descriptions given in terms of (pseudo) code, which is supposed to run locally on a node and contains guards over the local state (the received messages). For many cases, this formalization gap is easy to overcome, i.e., the translation from (pseudo) code to a threshold automaton is immediate, and can easily be done manually. However, for some algorithms this is not the case. Consider the consensus algorithm by Ben-Or given in Figure 3. The main challenge we faced in the formalization is to express the path that leads to the coin toss in line 16. Because it is in an "else" branch of an if statement with a "$\geq$ condition" over local variables, reaching the coin toss means that a local variable is a "$<$ condition". In other words, if we were to rewrite the pseudo code as a set of guarded commands, the coin toss would be guarded by a "$<$ condition" over receive variables. However, expressing the global constraints that can lead to the coin toss is captured by the rule r_{10}, given in Figure 5, which in fact represents a "$\geq$ condition" over global variables. This translation is non-trivial, and when done manually, requires intuition on the operation of the algorithm. As a result, in writing our benchmarks, we observed that when done manually, this translation is error-prone.

In [86, 87], we address the problem of automating the translation from pseudo code to a threshold automaton, for both asynchronous and synchronous threshold-guarded distributed algorithms. For randomized algorithms, whose local control flow motivated this line of work, the same results as for the asynchronous algorithms apply. In order to automate the translation, we need to formalize the local transition relation expressed by the pseudo code. To this end, we introduce a variant of threshold automata, called *receive threshold automata*, whose rules are guarded by expressions over the *local* receive variables.

Let $\mathsf{nr}_i(m)$ denote such a *receive variable* that encodes how many messages of type m process i has received and let $\mathsf{ns}(m)$ denote a *send variable*, that stores the number of sent messages of that type. Translating guards over receive variables $\mathsf{nr}_i(m)$ to guards over send variables $\mathsf{ns}(m)$, for each message type m, is based on *quantifier elimination* for Presburger arithmetic [75, 31, 76]. In order to obtain the most precise guards over the send variables, in the quantifier elimination step, the guards over the receive variables are strengthened by an *environment assumption* Env. As we will see in Section 6.2 below, the environment assumption encodes the relationship between the receive and send variables, which depends on the degree of synchrony and the fault model. Given a guard φ over the receive variables, a guard $\widehat{\varphi}$ over the send variables can be computed automatically by applying quantifier elimination to the formula $\varphi' \equiv \exists \mathsf{nr}_i(m_0) \dots \exists \mathsf{nr}_i(m_k)\ (\varphi \wedge \mathsf{Env})$, where $m_0, \dots, m_k$ are the message types that define the messages exchanged in the execution of the algorithm. This produces a quantifier-free formula $\widehat{\varphi}$ over the send variables.

The translation procedure was implemented in a prototype [86, 87] that automatically generates guards over the send variables, by using Z3 [33] to automate the quantifier elimination step. By applying the translation procedure to the guard of every rule in the receive threshold automaton given as input, a threshold automaton with no receive variables is obtained automatically. In [86, 87], it was shown that the translation procedure based on quantifier elimination is sound for both the asynchronous and synchronous case. This means that a system of n copies of an automatically generated threshold automaton over the send variables is an *overapproximation* of a system of n copies of the receive threshold automaton given as input. For a class of distributed algorithms that captures typical distributed algorithms found in the literature, it was also shown to be complete.

The translation procedure based on quantifier elimination thus closes the formalization gap between the original description of an algorithm (using received messages) and the threshold automaton of the algorithm, given as an input to a verification tool. More precisely, parameterized verification of threshold-guarded distributed algorithms, starting with a formal model of the pseudocode given by a receive threshold automaton, can be fully automated by: (i) automatically producing a formal model suitable for verification by applying the translation procedure based on quantifier elimination and (ii) automatically verifying its correctness by applying existing tools.

6.2. **Modeling Faults in Threshold Automata.** To model the behavior that the faults introduce, when producing a (receive) threshold automaton for a given algorithm, we have to capture the semantics of executing the code on a faulty process. To capture these semantics in the automaton, we typically need to introduce additional locations or rules, depending on the fault model. Also, depending on the fault model, we have different constraints on the values of the receive and send variables, which are encoded by an *environment assumption* Env (required to obtain the more precise guards after the quantifier elimination discussed above).

We consider two types of faults in this paper – crash and Byzantine faults. In the case of crash faults, a process may crash in the middle of a send-to-all operation, which results in the message being sent only to a subset of processes. In the case of Byzantine faults, no assumptions are made on the internal behavior of the faulty processes. That is, Byzantine-faulty processes may send any message in any order to any process or fail to send messages.

6.2.1. *Crash Faults.* Crash-faulty processes stop executing the algorithm prematurely and cannot restart. To model this behavior in a threshold automaton, we add so-called "crash" locations to which processes move from the "correct" locations. Processes that move to the "crash" locations remain there forever. In addition, we introduce send variables $\mathsf{ns}_f(m)$, for each message type m, that count the number of messages of type m sent by processes which are *crashing*, i.e., by processes that are moving from a "correct" to a "crashed" location.

The threshold automaton thus models the behavior of both correct and faulty processes explicitly. This allows us to express so-called *uniform* properties that also refer to states of faulty processes.[1] The environment assumption Env imposes constraints on the number of processes allowed to populate the "crash" locations, and on the number of received messages of each message type m. In particular, environment assumption Env requires that there are at most f processes in the "crash" locations, where f is the number of faulty processes. Additionally, for each message type m, the number of received messages of type m, stored in the receive variable $\mathsf{nr}_i(m)$ does not exceed the number of messages of type m sent by the correct and the crash-faulty processes, stored in the send variables $\mathsf{ns}(m)$ and $\mathsf{ns}_f(m)$, respectively.

In the synchronous case, where there exist strict guarantees on the message delivery, the environment assumption Env also bounds the value of the receive variables from below: the constraint $\mathsf{ns}(m) \leq \mathsf{nr}_i(m)$ encodes that all messages sent by correct processes are received in the round in which they are sent.

[1] For instance, in consensus "uniform agreement states that "no two processes decide on different values", while (non-uniform) "agreement" states that "no two *correct* processes decide on different values".

Table 3: Asynchronous fault-tolerant distributed algorithms that are verified by different generations of ByMC. For every technique and algorithm we show, whether the technique could verify the properties: safety (S), liveness (L), almost-sure termination under round-rigid aversaries (RRT), or none of them (-).

Algorithm	CA+SPIN[51]	CA+BDD[59]	CA+SAT[59]	SMT-S [56]	SMT-L [54]	SMT+MR[12]
FRB [27]	S+L	S+L	S	S	S+L	–
STRB [83]	S+L	S+L	S	S	S+L	–
ABA [19]	–	S+L	–	S	S+L	–
NBACG [46]	–	–	–	S	S+L	–
NBACR [77]	–	–	–	S	S+L	–
CBC [70]	–	–	–	S	S+L	–
CF1S [36]	–	S+L	–	S	S+L	–
C1CS [20]	–	–	–	S	S+L	–
BOSCO [81]	–	–	–	S	S+L	–
Ben-Or [8]	–	–	–	–	–	S+RRT
RABC [18]	–	–	–	–	–	S+RRT
kSet [69]	–	–	–	–	–	S+RRT
RS-BOSCO [81]	–	–	–	–	–	S+RRT

6.2.2. *Byzantine Faults.* To model the behavior of the Byzantine-faulty processes, which can act arbitrary, no new locations and rules are introduced in the threshold automaton. Instead, the threshold automaton is used to model the behavior of the correct processes, and the effect that the Byzantine-faulty processes have on the correct ones is captured in the guards and environment assumption. The number of messages sent by the Byzantine-faulty processes is overapproximated by the parameter f, which denotes the number of faults. That is, in the environment assumption Env, we have the constraint $\mathsf{nr}_i(m) \leq \mathsf{ns}(m) + f$, which captures that the number of received messages of type m does not exceed $\mathsf{ns}(m) + f$, which is an upper bound on the number of messages sent by the correct and Byzantine-faulty processes.

Since we do not introduce locations that explicitly model the behavior of the Byzantine-faulty processes, the threshold automaton is used to model the behavior of the $n - f$ correct processes only.[2] In addition to the above constraint that bounds the number of received messages from above, the environment assumption for the synchronous case also contains the constraint $\mathsf{ns}(m) \leq \mathsf{nr}_i(m)$. It is used to bound the number of received messages from below, and ensure that all messages sent by correct processes are received.

7. ByMC: Byzantine model checker

7.1. **Overview of the techniques implemented in ByMC.** Table 3 shows coverage of various asynchronous algorithms with the techniques that are implemented in ByMC. In the following, we give a brief description of these techniques.

[2] As classically no assumptions are made on the internals of Byzantine processes, it does not make sense to consider uniform properties. Thus we also do not need Byzantine faults explicit in the model.

Table 4: Synchronous fault-tolerant distributed algorithms verified with the bounded model checking approach from [84]. With ✓ we show that: the SMT based procedure finds a diameter bound with Z3 (**DIAM+Z3**) and CVC4 (**DIAM+CVC4**); there is a theoretical bound on the diameter (**DIAM+THM**). We verify safety (S) by bounded model checking with Z3 (**BMC+Z3**) and CVC4 (**BMC+CVC4**).

Algorithm	**DIAM+Z3**	**DIAM+CVC4**	**DIAM+THM**	**BMC+Z3**	**BMC+CVC4**
FloodSet [67]	✓	✓	–	S	S
FairCons [78]	✓	✓	–	S	S
PhaseKing [11]	✓	✓	–	S	S
PhaseQueen [10]	✓	✓	–	S	S
HybridKing [14]	✓	✓	–	S	S
ByzKing [14]	✓	✓	–	S	S
OmitKing [14]	✓	✓	–	S	S
HybridQueen [14]	–	–	–	–	–
ByzQueen [14]	✓	✓	–	S	S
OmitQueen [14]	✓	✓	–	S	S
FloodMin [67]	✓	✓	–	S	S
FloodMinOmit [14]	✓	✓	–	S	S
kSetOmit [78]	–	–	–	–	–
RB [83]	✓	✓	✓	S	S
HybridRB [14]	✓	✓	✓	S	S
OmitRB [14]	✓	✓	✓	S	S

We started the development of ByMC in 2012. We extended the classic $\{0, 1, \infty\}$-counter abstraction to threshold-guarded algorithms [51, 50, 45]. Instead of using the predefined intervals $[0, 1)$ and $[1, \infty)$, the tool was computing parametric intervals by simple static analysis, for instance, the intervals $[0, 1)$, $[1, t + 1)$, $[t + 1, n - t)$, and $[n - t, \infty)$. ByMC was automatically constructing the finite-state counter abstraction from protocol specifications in Parameterized Promela. This finite abstraction was automatically checked with Spin [49]. As this abstraction was typically too coarse for liveness checking, we have implemented a simple counterexample-guided abstraction refinement loop for parameterized systems. This technique is called CA+SPIN in Table 3.

Spin scaled only to two broadcast algorithms. Thus, we extended ByMC with the abstraction/checking loop that used nuXmv [25] instead of Spin. This technique is called CA+BDD in Table 3. Although this extension scaled better than CA+SPIN, we could only check two more benchmarks with it. Detailed discussions of the techniques CA+SPIN and CA+BDD can be found in [45, 57].

By running the abstraction/checking loop in nuXmv, we found that the bounded model checking algorithms of nuXmv could check long executions of our benchmarks. However, bounded model checking in general does not have completeness guarantees. In [55, 59], we have shown that the counter systems of (asynchronous) threshold automata have computable bounded diameters, which gave us a way to use bounded model checking as a complete verification approach for reachability properties. This technique is called CA+SAT in Table 3. Still, the computed upper bounds were too high for achieving complete verification.

The SMT-based techniques of Section 4 are called SMT-S (for safety) and SMT-L (for liveness) in Table 3. These techniques accept either threshold automata or Parametric

Promela on their input. As one can see, these techniques are the most efficient techniques that are implemented in ByMC. More details on the experiments can be found in the tool paper [58].

Finally, the technique for multi-round randomized algorithms is called SMT-MR in Table 3. This technique is explained in Section 5.

7.2. **Model checking synchronous threshold automata.** The bounded model checking approach for STA introduced in Section 3 is not yet integrated into ByMC. It is implemented as a stand-alone tool, available at [1]. In [84], we encoded multiple synchronous algorithms from the literature, such as consensus [67, 78, 11, 10, 14], k-set agreement (from [67], whose pseudocode is given in Figure 2 and [78]), and reliable broadcast (from [83, 14]) algorithms. We use Z3 [71] and CVC4 [7] as back-end SMT solvers. Table 4 gives an overview of the verified synchronous algorithms. For further details on the experimental results, see [84].

8. Towards verification of Tendermint consensus

Tendermint consensus is a fault-tolerant distributed algorithm for proof-of-stake blockchains [23]. Tendermint can handle Byzantine faults under the assumption of partial synchrony. It is running in the Cosmos network, where currently over 100 validator nodes are committing transactions and are managing the ATOM cryptocurrency [22].

8.1. **Challenges of verifying Tendermint.** Tendermint consensus heavily relies on threshold guards, as can be seen from its pseudo-code in [23][Algorithm 1]. For instance, one of the Tendermint rules has the following precondition:

$$\begin{aligned} &\textbf{upon}\,\langle \mathsf{PROPOSAL}, h_p, round_p, v, *\rangle\ \ \textbf{from}\ proposer(h_p, round_p)\\ &\qquad \textbf{AND}\ 2f+1\,\langle \mathsf{PREVOTE}, h_p, round_p, id(v)\rangle\\ &\qquad \textbf{while}\ valid(v) \wedge step_p \geq \mathsf{prevote}\ \text{for the first time} \end{aligned} \tag{8.1}$$

The rule (8.1) requires two kinds of messages: (1) a single message of type PROPOSAL carrying a proposal v from the process $proposer(h_p, round_p)$ that is identified by the current round $round_p$ and consensus instance h_p, and (2) messages of type PREVOTE from several nodes. Here the term $2F + 1$ (taken from the original paper) in fact does not refer to a number of processes. Rather, each process has a voting power (an integer that expresses how many votes a process has), and $2F+1$ (in combination with $N = 3T+1$) expresses that nodes that have sent PREVOTE have more than two-thirds of the voting power. Although this rule bears similarity with the rules of threshold automata, Tendermint consensus has the following features that cannot be directly modelled with threshold automata:

(1) In every consensus instance h_p and round $round_p$, a single proposer sends a value that the nodes vote on. The identity of the proposer can be accessed with the function $proposer(h_p, round_p)$. *This feature breaks symmetry among individual nodes*, which is required by our modelling with counter systems. Moreover, the proposer function should be fairly distributed among the nodes, e.g., it can be implemented with round robin.

(2) Whereas the classical example algorithms in this paper count messages, Tendermint evaluates the voting power of the nodes from which messages where received. This adds an additional layer of complexity.

(3) Liveness of Tendermint requires the distributed system to reach a global stabilization period, when every message could be delivered not later than after a bounded delay. This model of partial synchrony lies between synchronous and asynchronous computations and requires novel techniques for parameterized verification.

8.2. **Checking parameterized one-round safety with ByMC.** While we are not able to verify the complete Tendermint consensus algorithm in ByMC, we use ByMC to verify its one-round safety in the parameterized case. We do this in two steps. First, we take the TLA$^+$ specification [89] and manually abstract it, so the specification becomes structurally similar to a counter system of threshold automata. Second, we manually construct a threshold automaton from this TLA$^+$ specification[3].

We find this two-step approach easier to implement. TLA$^+$ is a general specification language [63], so it is much easier to write the first specification in TLA$^+$, rather than to write down a threshold automaton right away. Additionally, we first debugged our TLA$^+$ specification with the TLC model checker [92]. Once it worked for small values of the parameters, we did the translation to a threshold automaton. The TLA$^+$ specification can be found in Appendix A.

The following snippet encodes the rule (8.1) discussed above, and indeed is a threshold-guarded rule that encodes a transition from a process in location “prevote” to location “precommit”, provided enough “propose” and “prevote” messages are received:

$$\begin{array}{l}
\mathit{Line}36(v) \triangleq \\
\land \quad \mathit{nproposals}[v] > 0 \\
\land \quad \mathit{nprevotes}[v] + F \geq 2 * T + 1 \\
\land \quad \mathit{counters}[\text{“prevote”}] > 0 \\
\land \quad \mathit{nprecommits}' = [\mathit{nprecommits} \ \text{EXCEPT} \ ![v] = @ + 1] \\
\land \qquad \mathit{counters}' = [\mathit{counters} \ \text{EXCEPT} \qquad ! \qquad [\text{“prevote”}] = @ - 1, \ ! \\
[\text{“precommit”}] = @ + 1] \\
\land \quad \text{UNCHANGED} \ \langle \mathit{nproposals}, \mathit{nprevotes} \rangle
\end{array}$$

We translate the above rule into the following rules of a threshold automaton (an explanation of the syntax can found in [58]):

```
3: locPrevote -> locPrecommit
    when (nprop0 >= 1 && nprevote0 >= 2 * T + 1 - F)
    do {
      nprecommit0' == nprecommit0 + 1;
      nprecommitAll' == nprecommitAll + 1;
      unchanged(nprop0, nprop1,
                nprevote0, nprevote1, nprevoteNil, nprevoteAll,
                nprecommit1, nprecommitNil);
    };
4: locPrevote -> locPrecommit
    when (nprop1 >= 1 && nprevote1 >= 2 * T + 1 - F)
    do {
      nprecommit1' == nprecommit1 + 1;
      nprecommitAll' == nprecommitAll + 1;
```

[3] The TLA$^+$ specification and the threshold automaton are publicly available at: https://github.com/konnov/fault-tolerant-benchmarks/tree/master/lmcs20.

```
  unchanged(nprop0, nprop1,
            nprevote0, nprevote1, nprevoteNil, nprevoteAll,
            nprecommit0, nprecommitNil);
};
```

The standard TLA$^+$ model checker TLC can check the specification for $N \leq 20$ in reasonable time. We checked the safety of consensus in one round: No two correct processes decide differently in the same round. As TLC is an explicit-state model checker, it did not scale to the value of $N = 125$, which is the current number of validators in the Cosmos main chain. In contrast, we have verified this property with ByMC in the parameterized case in 2 seconds.

In this model of the algorithm, the one-round safety is satisfied under the assumption of $N = 3 \cdot T + 1 \wedge T \geq F$. When we change the assumption to either $N \geq 3 \cdot T + 1 \wedge T \geq F$, or $N = 3 \cdot T + 1$, ByMC produces counterexamples to the property. The actual implementation of Tendermint is different, as it dynamically recomputes the voting powers of the validators and thus it recomputes the thresholds. However, we follow the assumptions of [23] in our modeling.

We observe that our manual abstraction of the TLA$^+$ specification into a threshold automaton is rather mechanical. This abstraction could be done in a fully automatic pipeline: From TLA$^+$ to a receive threshold automaton, and from a receive threshold automaton to a threshold automaton (see Section 6.1). We are planning to use automatic abstractions to build a bridge between ByMC and Apalache — a symbolic model checker for TLA$^+$ [52].

8.3. Open problems for parameterized verification of multi-round safety.

To verify multi-round safety of Tendermint, we would like to invoke a reduction argument similar to the one explained in Section 5.2. However, Tendermint contains the following rule that prevents us from directly applying the reduction result:

$$\begin{array}{l} \textbf{upon}\ \langle \mathsf{PROPOSAL}, h_p, \mathit{round}_p, v, \mathit{vr} \rangle\ \ \textbf{from}\ \mathit{proposer}(h_p, \mathit{round}_p) \\ \qquad \textbf{AND}\ 2f+1\ \langle \mathsf{PREVOTE}, h_p, \mathit{vr}, \mathit{id}(v) \rangle \\ \qquad \textbf{while}\ \mathit{step}_p = \mathsf{propose} \wedge (\mathit{vr} \geq 0 \wedge \mathit{vr} < \mathit{round}_p) \\ \qquad \ldots \end{array} \tag{8.2}$$

The rule in Equation (8.2) allows a process to make a step by using messages from a past round vr. As a result, Tendermint is not communication-closed [40, 28, 32]. Extending the reduction argument to multi-round systems that are not communication closed is subject to our ongoing work.

9. Conclusions

Practical approaches to computer-aided verification of distributed algorithms and systems is a lively research area as well: Approaches range from mechanized verification [48, 90, 80] over deductive verification [35, 9, 73, 38, 32] to automated techniques [17, 60, 5, 44]. In our work, we follow the idea of identifying fragments of automata and logic that are sufficiently expressive for capturing interesting algorithms and specifications, as well as amenable for completely automated verification. We introduced threshold automata for that and implemented our verification techniques in the open source tool ByMC [58]. By doing so, we verified several challenging distributed algorithms; most of them were verified for the first time.

The threshold automata framework has proved to be both of practical relevance as well as of theoretical interest. There are several ongoing projects that consider automatic generation of threshold automata from code, complexity theoretic analysis of verification problems, and more refined probabilistic reasoning. The restrictions posed on the form of allowed threshold guards and the corresponding actions, are inspired by the typical forms seen in the benchmarks. The work presented in [61] explores various relaxations of the standard restrictions, such as non-linear threshold guards, guards that compare shared variables, decrementing shared variables, or incrementing them inside self-loops. For each of these theoretical extensions, the authors investigate the existence of a bounded diameter and decidability of reachability properties. For the standard setting, a systematic analysis of computational complexity of verification and synthesis in threshold automata has been conducted in [6]. The authors express the reachability relation as a formula in existential Presburger arithmetic, and therefore prove that coverability and reachability problems, as well as model checking of $\mathsf{ELTL}_{\mathsf{FT}}$ properties are NP-complete, while synthesizing threshold guards is Σ_p^2-complete. An extension of the work from Section 5 has been explored in [13], where the setting of the round-rigid adversaries has been lifted to a broader domain, namely, to the more natural weak adversaries. The paper introduces a new (threshold automata-based) modeling that distinguishes individual processes, and it presents a reduction theorem claiming that for every weak adversary there exists a round-rigid one with the same properties. This implies that the verification results from Section 5 and the lower part of Table 3 hold under a wider class of adversaries than claimed.

Acknowledgments. This survey is based on the results of a long-lasting research agenda [51, 53, 54, 64, 12, 85]. We are grateful to our past and present collaborators Nathalie Bertrand, Roderick Bloem, Annu Gmeiner, Jure Kukovec, Ulrich Schmid, Helmut Veith, and Florian Zuleger, who contributed to many of the described ideas that are now implemented in ByMC.

References

[1] Bounded Model Checking of STA. https://github.com/istoilkovska/syncTA

[2] Abraham, I., Malkhi, D., Nayak, K., Ren, L., Spiegelman, A.: Solidus: An incentive-compatible cryptocurrency based on permissionless Byzantine consensus. CoRR abs/1612.02916 (2016), http://arxiv.org/abs/1612.02916

[3] Aguilera, M., Toueg, S.: The correctness proof of Ben-Or's randomized consensus algorithm. Distributed Computing pp. 1–11 (2012)

[4] Apt, K., Kozen, D.: Limits for automatic verification of finite-state concurrent systems. IPL 15, 307–309 (1986)

[5] Bakst, A., von Gleissenthall, K., Kici, R.G., Jhala, R.: Verifying distributed programs via canonical sequentialization. PACMPL 1(OOPSLA), 110:1–110:27 (2017)

[6] Balasubramanian, A.R., Esparza, J., Lazić, M.: Complexity of verification and synthesis of threshold automata. In: ATVA. Lecture Notes in Computer Science, vol. 12302, pp. 144–160. Springer (2020)

[7] Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV. pp. 171–177 (2011)

[8] Ben-Or, M.: Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In: PODC. pp. 27–30 (1983)

[9] Berkovits, I., Lazic, M., Losa, G., Padon, O., Shoham, S.: Verification of threshold-based distributed algorithms by decomposition to decidable logics. In: CAV. LNCS, vol. 11562, pp. 245–266. Springer (2019)

[10] Berman, P., Garay, J.A., Perry, K.J.: Asymptotically Optimal Distributed Consensus. Tech. rep., Bell Labs (1989), `plan9.bell-labs.co/who/garay/asopt.ps`

[11] Berman, P., Garay, J.A., Perry, K.J.: Towards Optimal Distributed Consensus (Extended Abstract). In: FOCS. pp. 410–415 (1989)

[12] Bertrand, N., Konnov, I., Lazic, M., Widder, J.: Verification of Randomized Consensus Algorithms Under Round-Rigid Adversaries. In: CONCUR 2019. LIPIcs, vol. 140, pp. 33:1–33:15 (2019)

[13] Bertrand, N., Lazić, M., Widder, J.: A reduction theorem for randomized distributed algorithms under weak adversaries. In: VMCAI (2021), (to appear)

[14] Biely, M., Schmid, U., Weiss, B.: Synchronous Consensus Under Hybrid Process and Link Failures. Theor. Comput. Sci. 412(40), 5602–5630 (2011)

[15] Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic Model Checking without BDDs. In: TACAS. LNCS, vol. 1579, pp. 193–207 (1999)

[16] Bloem, R., Jacobs, S., Khalimov, A., Konnov, I., Rubin, S., Veith, H., Widder, J.: Decidability of Parameterized Verification. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool (2015)

[17] Bouajjani, A., Enea, C., Ji, K., Qadeer, S.: On the completeness of verifying message passing programs under bounded asynchrony. In: CAV. pp. 372–391 (2018)

[18] Bracha, G.: Asynchronous Byzantine agreement protocols. Inf. Comput. 75(2), 130–143 (1987)

[19] Bracha, G., Toueg, S.: Asynchronous consensus and broadcast protocols. J. ACM 32(4), 824–840 (1985)

[20] Brasileiro, F.V., Greve, F., Mostéfaoui, A., Raynal, M.: Consensus in one communication step. In: PaCT. LNCS, vol. 2127, pp. 42–50 (2001)

[21] Buchman, E.: Tendermint: Byzantine Fault Tolerance in the Age of Blockchains. Master's thesis, University of Guelph (2016), `http://hdl.handle.net/10214/9769`

[22] Buchman, E., Kwon, J.: Cosmos whitepaper: a network of distributed ledgers (2018), `https://cosmos.network/resources/whitepaper`

[23] Buchman, E., Kwon, J., Milosevic, Z.: The latest gossip on bft consensus. arXiv preprint arXiv:1807.04938 (2018), `https://arxiv.org/abs/1807.04938`

[24] Buterin, V.: A next-generation smart contract and decentralized application platform (2014)

[25] Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuxmv symbolic model checker. In: CAV. pp. 334–342 (2014)

[26] Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM 43(2), 225–267 (1996)

[27] Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. JACM 43(2), 225–267 (March 1996)

[28] Chaouch-Saad, M., Charron-Bost, B., Merz, S.: A reduction theorem for the verification of round-based distributed algorithms. In: RP. pp. 93–106 (2009), `http://dx.doi.org/10.1007/978-3-642-04420-5_10`

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

[30] Chaudhuri, S., Herlihy, M., Lynch, N.A., Tuttle, M.R.: Tight Bounds for k-set Agreement. J. ACM 47(5), 912–943 (2000)

[31] Cooper, D.C.: Theorem proving in arithmetic without multiplication. Machine intelligence 7(91-99) (1972)

[32] Damian, A., Drăgoi, C., Militaru, A., Widder, J.: Communication-closed asynchronous protocols. In: CAV. pp. 344–363 (2019)

[33] De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 1579, pp. 337–340 (2008)

[34] Decker, C., Seidel, J., Wattenhofer, R.: Bitcoin meets strong consistency. In: ICDCN. pp. 13:1–13:10 (2016), `https://doi.org/10.1145/2833312.2833321`

[35] Desai, A., Garg, P., Madhusudan, P.: Natural proofs for asynchronous programs using almost-synchronous reductions. In: OOPSLA. pp. 709–725 (2014)

[36] Dobre, D., Suri, N.: One-step consensus with zero-degradation. In: DSN. pp. 137–146 (2006)

[37] Dolev, D., Dwork, C., Stockmeyer, L.: On the minimal synchronism needed for distributed consensus. J. ACM 34, 77–97 (1987)

[38] Drăgoi, C., Henzinger, T.A., Veith, H., Widder, J., Zufferey, D.: A Logic-Based Framework for Verifying Consensus Algorithms. In: VMCAI. LNCS, vol. 8318, pp. 161–181 (2014)

[39] Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J.ACM 35(2), 288–323 (1988)

[40] Elrad, T., Francez, N.: Decomposition of distributed programs into communication-closed layers. Sci. Comput. Program. 2(3), 155–173 (1982)

[41] Emerson, E., Namjoshi, K.: Reasoning about rings. In: POPL. pp. 85–94 (1995)

[42] Esparza, J.: Decidability of model checking for infinite-state concurrent systems. Acta Informatica 34(2), 85–107 (1997)

[43] Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM 32(2), 374–382 (1985)

[44] v. Gleissenthall, K., Gökhan Kici, R., Bakst, A., Stefan, D., Jhala, R.: Pretend synchrony. In: POPL (2019), (to appear)

[45] Gmeiner, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Tutorial on parameterized model checking of fault-tolerant distributed algorithms. In: Formal Methods for Executable Software Models. pp. 122–171. LNCS, Springer (2014)

[46] Guerraoui, R.: Non-blocking atomic commit in asynchronous distributed systems with failure detectors. Distributed Computing 15(1), 17–25 (2002)

[47] Hadzilacos, V., Toueg, S.: Fault-tolerant broadcasts and related problems. In: Mullender, S. (ed.) Distributed systems (2nd Ed.). pp. 97–145 (1993)

[48] Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: Ironfleet: proving safety and liveness of practical distributed systems. Commun. ACM 60(7), 83–92 (2017)

[49] Holzmann, G.: The SPIN Model Checker. Addison-Wesley (2003)

[50] John, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Counter Attack on Byzantine Generals: Parameterized Model Checking of Fault-tolerant Distributed Algorithms (October 2012), http://arxiv.org/abs/1210.3846

[51] 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)

[52] Konnov, I., Kukovec, J., Tran, T.: TLA+ model checking made symbolic. PACMPL 3(OOPSLA), 123:1–123:30 (2019)

[53] Konnov, I., Lazić, M., Veith, H., Widder, J.: Para^2: Parameterized path reduction, acceleration, and SMT for reachability in threshold-guarded distributed algorithms. Formal Methods in System Design 51(2), 270–307 (2017)

[54] Konnov, I., Lazić, M., Veith, H., Widder, J.: A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In: POPL. pp. 719–734 (2017)

[55] Konnov, I., Veith, H., Widder, J.: On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. In: CONCUR. LNCS, vol. 8704, pp. 125–140 (2014)

[56] Konnov, I., Veith, H., Widder, J.: SMT and POR beat counter abstraction: Parameterized model checking of threshold-based distributed algorithms. In: CAV (Part I). LNCS, vol. 9206, pp. 85–102 (2015)

[57] Konnov, I., Veith, H., Widder, J.: What you always wanted to know about model checking of fault-tolerant distributed algorithms. In: PSI 2015, in Memory of Helmut Veith, Revised Selected Papers. LNCS, vol. 9609, pp. 6–21. Springer (2016)

[58] Konnov, I., Widder, J.: ByMC: Byzantine model checker. In: Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems. pp. 327–342. Springer International Publishing, Cham (2018), https://hal.inria.fr/hal-01909653

[59] Konnov, I.V., Veith, H., Widder, J.: On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. Information and Computation 252, 95–109 (2017)

[60] Kragl, B., Qadeer, S., Henzinger, T.A.: Synchronizing the asynchronous. In: CONCUR. pp. 21:1–21:17 (2018)

[61] Kukovec, J., Konnov, I., Widder, J.: Reachability in parameterized systems: All flavors of threshold automata. In: CONCUR. LIPIcs, vol. 118, pp. 19:1–19:17 (2018)

[62] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)

[63] Lamport, L.: Specifying systems: The TLA+ language and tools for hardware and software engineers. Addison-Wesley (2002)

[64] Lazić, M., Konnov, I., Widder, J., Bloem, R.: Synthesis of distributed algorithms with parameterized threshold guards. In: OPODIS. LIPIcs, vol. 95, pp. 32:1–32:20 (2017), https://doi.org/10.4230/LIPIcs.OPODIS.2017.32

[65] Le Lann, G.: Distributed systems – towards a formal approach. In: IFIP Congress. pp. 155–160 (1977), http://www-roc.inria.fr/novaltis/publications/IFIP%20Congress%201977.pdf

[66] Lincoln, P., Rushby, J.: A formally verified algorithm for interactive consistency under a hybrid fault model. In: FTCS. pp. 402–411 (1993)

[67] Lynch, N.: Distributed Algorithms. Morgan Kaufman, San Francisco, USA (1996)

[68] Malekpour, M.R., Siminiceanu, R.: Comments on the "Byzantine self-stabilizing pulse synchronization". protocol: Counterexamples. Tech. rep., NASA (Feb 2006), http://shemesh.larc.nasa.gov/fm/papers/Malekpour-2006-tm213951.pdf

[69] Mostéfaoui, A., Moumen, H., Raynal, M.: Randomized k-set agreement in crash-prone and Byzantine asynchronous systems. Theor. Comput. Sci. 709, 80–97 (2018)

[70] Mostéfaoui, A., Mourgaya, E., Parvédy, P.R., Raynal, M.: Evaluating the condition-based approach to solve consensus. In: DSN. pp. 541–550 (2003)

[71] de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS. pp. 337–340 (2008)

[72] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008), https://bitcoin.org/bitcoin.pdf

[73] Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. In: PLDI. pp. 614–630 (2016)

[74] Pease, M.C., Shostak, R.E., Lamport, L.: Reaching agreement in the presence of faults. J. ACM 27(2), 228–234 (1980)

[75] Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. Comptes Rendus du I congres de Mathématiciens des Pays Slaves (1929)

[76] Pugh, W.: A practical algorithm for exact array dependence analysis. Communications of the ACM 35(8) (1992)

[77] Raynal, M.: A case study of agreement problems in distributed systems: Non-blocking atomic commitment. In: HASE. pp. 209–214 (1997)

[78] Raynal, M.: Fault-Tolerant Agreement in Synchronous Message-Passing Systems. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers (2010)

[79] Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Comput. Surv. 22(4), 299–319 (1990)

[80] Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and proving with distributed protocols. PACMPL 2(POPL), 28:1–28:30 (2018)

[81] Song, Y.J., van Renesse, R.: Bosco: One-step Byzantine asynchronous consensus. In: DISC. LNCS, vol. 5218, pp. 438–450 (2008)

[82] Srikanth, T.K., Toueg, S.: Optimal clock synchronization. Journal of the ACM 34(3), 626–645 (1987)

[83] Srikanth, T., Toueg, S.: Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. Dist. Comp. 2, 80–94 (1987)

[84] Stoilkovska, I., Konnov, I., Widder, J., Zuleger, F.: Verifying safety of synchronous fault-tolerant algorithms by bounded model checking. In: TACAS. LNCS, vol. 11428, pp. 357–374. Springer (2019)

[85] Stoilkovska, I., Konnov, I., Widder, J., Zuleger, F.: Verifying Safety of Synchronous Fault-Tolerant Algorithms by Bounded Model Checking. In: TACAS. LNCS, vol. 11428, pp. 357–374 (2019)

[86] Stoilkovska, I., Konnov, I., Widder, J., Zuleger, F.: Eliminating message counters in threshold automata. In: ATVA. Lecture Notes in Computer Science, vol. 12302, pp. 196–212. Springer (2020)

[87] Stoilkovska, I., Konnov, I., Widder, J., Zuleger, F.: Eliminating message counters in synchronous threshold automata. In: VMCAI (2021), (to appear)

[88] Suzuki, I.: Proving properties of a ring of finite-state machines. Inf. Process. Lett. 28(4), 213–214 (1988)

[89] Systems, I.: Tla+ specification of tendermint safety and accountability (2020), https://github.com/informalsystems/tendermint-rs/tree/master/docs/spec/tendermint-fork-cases

[90] 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)

[91] Yin, M., Malkhi, D., Reiter, M.K., Golan-Gueta, G., Abraham, I.: Hotstuff: BFT consensus with linearity and responsiveness. In: PODC. pp. 347–356 (2019)

[92] Yu, Y., Manolios, P., Lamport, L.: Model checking TLA$^+$ specifications. In: Correct Hardware Design and Verification Methods, pp. 54–66. Springer (1999)

Appendix A. Abstraction of Tendermint Consensus Round in TLA+

```
──────────────── MODULE Tendermint_1round_safety ────────────────
 *
 * This is a very simplified version of Tendermint that is tuned for safety checking
 * with counter systems :
 *
 * The simplifications are as follows :
 * − the model is completely asynchronous, as we are only concerned with safety
 * − we only consider binary consensus
 * − we only consider one height and one round
 * − as we are modeling only one round, there is no locking mechanism in the specification
 * − we use symmetry arguments to replace sets of messages with message counters
 * − the proposer is modeled as a non-deterministic assignment in the initial state
 * − there are no hashes and no validity checks
 *
 * A more complete specification for multiple rounds can be found at :
 *
      *      https://github.com/informalsystems/tendermint-rs/blob/master/docs/spec/tendermint-fork-
cases/TendermintAcc3.tla
 *
 * The original specification of Tendermint can be found at :
 *
 * https://arxiv.org/abs/1807.04938
 *
 * Igor Konnov, Marijana Lazic, Ilina Stoilkovska, Josef Widder, 2020
EXTENDS Integers

CONSTANTS N,        the number of processes in the system
          T,        the threshold on the number of faults
          F         the number of actual Byzantine faults

    the Tendermint algorithm is specified under this assumption
ASSUME (N = 3 ∗ T + 1 ∧ F ≤ T ∧ F ≥ 0)

Locs ≜ {
   "propose", "prevote", "precommit", "decide0", "decide1", "nodecision"
}

Nil ≜ "nil"
Values ≜ {"0", "1"}                    values 0 and 1
ValuesExt ≜ Values ∪ {Nil}

VARIABLES
   counters,       the counter for every location: a function from Locs to Naturals
   nproposals,       the numbers of proposals sent around, for values 0 and 1
   nprevotes,        the numbers of prevotes sent around, for values 0, 1, and Nil
   nprecommits       the numbers of prevotes sent around, for value 0, 1, and Nil

Init ≜
        initially, all but Byzantine processes are in the "propose" state
   ∧ counters = [l ∈ Locs ↦ IF l = "propose" THEN N − F ELSE 0]
```

```
            the proposer can send 0 or 1 (or both, when it is faulty)
    ∧ nproposals ∈ [Values → {0, 1}]
    ∧ nprevotes = [v ∈ ValuesExt ↦ 0]
    ∧ nprecommits = [v ∈ ValuesExt ↦ 0]

        The action in line 22.
        Line 28 is the same as Line22 in our safety abstraction.
Line22(v) ≜
    ∧ nproposals[v] > 0
    ∧ counters["propose"] > 0
    ∧ ∨ nprevotes′ = [nprevotes EXCEPT ![v] = @ + 1]            line 24
      ∨ nprevotes′ = [nprevotes EXCEPT ![Nil] = @ + 1]          line 26
    ∧ counters′ = [counters EXCEPT !["propose"] = @ − 1,
                                   !["prevote"]  = @ + 1]
    ∧ UNCHANGED ⟨nproposals, nprecommits⟩

Line36(v) ≜
            note that line 36 also works for step_p = precommit,
            but it effectively does nothing in our safety abstraction.
    ∧ nproposals[v] > 0
    ∧ nprevotes[v] + F ≥ 2 ∗ T + 1
    ∧ counters["prevote"] > 0
    ∧ nprecommits′ = [nprecommits EXCEPT ![v] = @ + 1]
    ∧ counters′ = [counters EXCEPT !["prevote"] = @ − 1,
                                   !["precommit"] = @ + 1]
    ∧ UNCHANGED ⟨nproposals, nprevotes⟩

Line44 ≜
    ∧ nprevotes[Nil] + F ≥ 2 ∗ T + 1
    ∧ counters["prevote"] > 0
    ∧ nprecommits′ = [nprecommits EXCEPT ![Nil] = @ + 1]
    ∧ counters′ = [counters EXCEPT !["prevote"] = @ − 1,
                                   !["precommit"] = @ + 1]
    ∧ UNCHANGED ⟨nproposals, nprevotes⟩

Line49(v) ≜
    ∧ nproposals[v] > 0
    ∧ nprecommits[v] + F ≥ 2 ∗ T + 1
    ∧ ∃ loc ∈ {"propose", "prevote", "precommit"} :
      LET decision ≜
           IF v = "0" THEN "decide0" ELSE "decide1"
      IN
       ∧ counters[loc] > 0
       ∧ counters′ = [counters EXCEPT ![loc] = @ − 1,
                                      ![decision] = @ + 1]
       ∧ UNCHANGED ⟨nproposals, nprevotes, nprecommits⟩

OnTimeoutPropose ≜
    ∧ counters["propose"] > 0
    $\land\ nprevotes' = [nprevotes\ \text{EXCEPT}\ ![Nil] = @ + 1]$
    $\land\ counters' = [counters\ \text{EXCEPT}\ ![\text{“propose”}] = @ - 1,$
                                            $![\text{“prevote”}] = @ + 1]$
    $\land\ \text{UNCHANGED}\ \langle nproposals,\ nprecommits \rangle$

$Line34OnTimeoutPrevote \triangleq$
    $\land\ nprevotes[\text{“0”}] + nprevotes[\text{“1”}] + nprevotes[Nil] + F \geq 2 * T + 1$
    $\land\ counters[\text{“prevote”}] > 0$
    $\land\ nprecommits' = [nprecommits\ \text{EXCEPT}\ ![Nil] = @ + 1]$
    $\land\ counters' = [counters\ \text{EXCEPT}\ ![\text{“prevote”}] = @ - 1,$
                                            $![\text{“precommit”}] = @ + 1]$
    $\land\ \text{UNCHANGED}\ \langle nproposals,\ nprevotes \rangle$

$Line47OnTimeoutPrecommit \triangleq$
    $\land\ nprecommits[\text{“0”}] + nprecommits[\text{“1”}] + nprecommits[Nil] + F \geq 2 * T + 1$
    $\land\ counters[\text{“precommit”}] > 0$
    $\land\ counters' = [counters\ \text{EXCEPT}\ ![\text{“precommit”}] = @ - 1,$
                                            $![\text{“nodecision”}] = @ + 1]$
    $\land\ \text{UNCHANGED}\ \langle nproposals,\ nprevotes,\ nprecommits \rangle$

$Next \triangleq$
    $\lor\ \exists\, v \in Values : Line22(v)$
    $\lor\ \exists\, v \in Values : Line36(v)$
    $\lor\ \exists\, v \in Values : Line44$
    $\lor\ \exists\, v \in Values : Line49(v)$
    $\lor\ OnTimeoutPropose$
    $\lor\ Line34OnTimeoutPrevote$
    $\lor\ Line47OnTimeoutPrecommit$
    $\lor\ \text{UNCHANGED}\ \langle counters,\ nproposals,\ nprevotes,\ nprecommits \rangle$

$RoundAgreementInv \triangleq$
    $counters[\text{“decide0”}] = 0 \lor counters[\text{“decide1”}] = 0$
```