ByMC: Byzantine Model Checker

Igor Konnov, Josef Widder

ByMC: Byzantine Model Checker

Igor Konnov[1] and Josef Widder[2] *

[1] University of Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
igor.konnov@inria.fr
[2] TU Wien (Vienna University of Technology), Vienna, Austria
widder@forsyte.at

Abstract. In recent work [12,10], we have introduced a technique for automatic verification of *threshold-guarded distributed algorithms* that have the following features: (1) up to t of processes may crash or behave Byzantine; (2) the correct processes count messages and 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., $n > 3t$.
In this paper, we present Byzantine Model Checker that implements the above-mentioned technique. It takes two kinds of inputs, namely, (i) threshold automata (the framework of our verification techniques) or (ii) Parametric Promela (which is similar to the way in which the distributed algorithms were described in the literature).
We introduce a *parallel* extension of the tool, which exploits the parallelism enabled by our technique on an MPI cluster. We compare performance of the original technique and of the extensions by verifying 10 benchmarks that model fault-tolerant distributed algorithms from the literature. For each benchmark algorithm we check two encodings: a manual encoding in threshold automata vs. a Promela encoding.

1 Introduction

In recent work [11,12,10] we applied bounded model checking to verify reachability properties of threshold-based fault-tolerant distributed algorithms (FTDA), which are parameterized in the number of processes n and the fraction of faults t. FTDAs typically work only under arithmetic resilience conditions such as $n > 3t$. Our methods allow us to do parameterized verification of sophisticated FTDAs [18,20,6,21,3,5] that have not been automatically verified before. Our bounded model checking technique produces a number of queries to a Satisfiability Modulo Theories solver (SMT). These queries correspond to different execution patterns.

* Supported by: the Vienna Science and Technology Fund (WWTF) grant APALACHE (ICT15-103); and the Austrian Science Fund (FWF) through the National Research Network RiSE (S11403 and S11405), and project PRAVDA (P27722). The computational results presented have been achieved [in part] using the Vienna Scientific Cluster (VSC).

```
1  // n processes follow the code:
2  input u_i ∈ {0, 1};
3  send u_i to all;
4  wait until some value v_i ∈ {0, 1}
5    is received ⌈(n+1)/2⌉ times;
6  decide on v_j;
```

Fig. 1. Naïve Voting algorithm

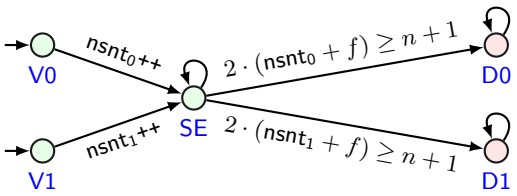

Fig. 2. A threshold automaton for Naïve Voting

In [12], we conjectured that, by design, this technique allows many SMT queries to be checked in parallel. In this paper, we present a parallel extension of ByMC that executes SMT queries in a computer cluster.

The contributions of this paper are as follows:

1. We present the tool ByMC 2.4.1 that implements sequential and parallel verification [12,10]. The parallel verification is implemented with MPI (Message Passing Interface).
2. We introduce the details of the parallel extension of the technique and perform experimental evaluation, both for the sequential and parallel versions of the tool.
3. We report the experimental results both for the abstractions that are automatically constructed from Promela code (as in [12,10]) and for manual abstractions in terms of threshold automata, which we use as a direct input for the first time. Our experiments show that explicit modeling of fault-tolerant distributed algorithms with threshold automata leads to a dramatic speed up in most cases.

2 Distributed algorithm example: Naïve Voting

In order to describe what kind of distributed algorithms our tool ByMC is designed for, we start with a simple threshold-guarded algorithm. In this section, we take the point of view of an algorithm designer and apply the arguments that can be found in the distributed algorithms literature [1,16]. Consider a distributed system of n processes, whose goal is to unanimously decide on a binary value $v \in \{0, 1\}$. We would like to design a distributed algorithm that satisfies the following three properties[3]:

- *Agreement.* No two correct processes decide on different values (0 and 1).
- *Validity.* If a correct process decides on a value $v \in \{0, 1\}$, then there is a process i, whose initial value u_i equals v.
- *Termination.* All correct processes eventually decide.

Figure 1 shows a naïve attempt to solve this problem by majority voting. As usual in the distributed algorithms literature, we give a solution in pseudo-code, which is supposed to work as follows. Each process starts with a binary

[3] Agreement, validity, and termination are typical properties of consensus [1,16].

value $u_i \in \{0, 1\}$ and sends u_i to all processes, including itself. When a process receives a value $v \in \{0, 1\}$ from a majority of processes, it decides on v.

Does Naïve Voting satisfy agreement, validity, and termination? Unfortunately, the pseudo-code does not provide us with sufficient detail to answer the question: Assumptions about the process scheduler, message-delivery, possible faults, etc. are missing. For instance, if messages can be lost, a process may never receive sufficiently many messages to get over the guard in line 4. Thus we have to specify *systems assumptions*. Let us consider an asynchronous model [1,16] with crashes and Byzantine faults [19]:

- *Asynchronous computations.* Every correct process is scheduled infinitely often, and there are no assumptions on the relative processor speeds. The process steps are interleaved.
- *Reliable communication.* The processes communicate via message passing. Every message sent by a correct process is eventually delivered, although there are no timing or ordering assumptions about message delivery.
- *Faults.* A fraction f of processes may fail. For instance, they can crash or behave Byzantine — the faulty processes do not follow the algorithm. There is an upper bound $t \geq f$ on the number of faults. We assume $n > 3t$ for the Byzantine faults, and $n > 2t$ for the crash faults.

Manual proofs. Below, we manually reason about the algorithm's correctness. Such proofs are common in the distributed algorithms literature, cf. [22,21,24].

Validity We consider the Byzantine case here, which is more complicated. In order to decide on a value v in line 6, a correct process has to receive $\lceil \frac{n+1}{2} \rceil$ messages carrying v. By the assumption on the number of faults ($n > 3t$ and $t \geq f$), we have $f < \lceil \frac{n+1}{2} \rceil$, and if a process decides on v in 6, there is at least one correct process that has sent the value v in line 3. Thus, the algorithm satisfies "Validity".

Agreement Whether the algorithm satisfies "Agreement" depends on the considered fault model:

- *No faults or crash faults.* By line 4, a process has to receive the same value from $\lceil \frac{n+1}{2} \rceil$ distinct processes. Since $2 \cdot \lceil \frac{n+1}{2} \rceil > n$, and each process sends only one value (line 3), no two processes $i, j : \ 1 \leq i < j \leq n$ can reach line 6 with different values $v_i \neq v_j$. Thus, the processes cannot decide differently, and agreement is satisfied.
- *Byzantine faults.* When $f > 0$, the Byzantine processes can send value 0 to a process i and value 1 to a process $j : j \neq i$. If the initial states of the correct processes are split into two equal sets, that is, $n - f = 2 \cdot |\{k \in \{1 \dots n\} \colon k \text{ is correct and } u_k = 0\}|$, then the processes i and j reach line 6 with the values $v_i = 0$ and $v_j = 1$. As a result, agreement can be violated, and a verification tool must produce a counterexample.

Termination Assume that there are no faults ($f = 0$) and the initial states are equally partitioned, that is, $n = 2 \cdot |\{k \in \{1 \dots n\} \colon k \text{ is correct and } u_k = 0\}|$. No process can pass beyond line 4, as none of the initial value sets form a majority. Therefore, Naïve Voting violates liveness, namely, "Termination".

This subtle bug renders the algorithm useless! A tool should thus not only check invariants, but also find counterexamples to liveness specifications.

The manual proofs are tricky, as they combine several kinds of reasoning: temporal reasoning, local reasoning about process code, global reasoning about the number of messages, correct and faulty processes, etc. Our tool ByMC automatically proves temporal properties (or finds counterexamples) of distributed algorithms that (i) communicate by sending to all, and (ii) contain actions that are guarded by comparison of the number of received messages against a linear combination of parameter values (e.g., for a majority).

3 Inputs: Parametric Promela & Threshold Automata

The algorithm in Figure 1 looks quite simple. However, as one can see from the assumptions on, e.g., faults and communication in Section 2, many details (that are often deemed "non-essential" by algorithm designers) are missing in the pseudo code. Our tool addresses this challenge by supporting two formal languages that are tailored for modeling of threshold-guarded distributed algorithms and the system assumptions: parametric Promela [9,8] and threshold automata [11]. Parametric Promela offers modeling that closely mimicks the behavior of the pseudo code statements, whereas threshold automata are an abstraction that allows for efficient model checking techniques [12,10]. When given code in parametric Promela, ByMC internally applies data abstraction to construct a threshold automaton, as explained in [13]. However, the automatically computed threshold automata are usually much larger than those constructed manually by a distributed algorithms expert. For this reason, the user can directly give a threshold automaton as the input to the tool.

3.1 Parametric Promela

Promela is the input language of the Spin model checker [7]. As it is designed to specify concurrent systems, several features are suitable for capturing distributed algorithms. However, Spin is a finite state model checker, and so Promela only allows us to specify finite state systems. We have thus extended Promela in order to have a parametric number of processes and faults, etc. In the following we will discuss some of our extensions.

Figure 3 shows a model of the Naïve Voting algorithm from Figure 1. This example contains all the essential features of parametric Promela. In line 2, we declare parameters: the number of processes n, the number of Byzantine processes f, and the minimal size of a majority set, that is, $\lceil \frac{n+1}{2} \rceil$. In line 3, we declare two shared integer variables `nsnt0` and `nsnt1` that store the number of zeroes and ones sent by the *correct* processes. The expressions `assume(...)` in lines 4–5 restrict the choice of parameter values.

The behavior of the $n - f$ correct processes is modeled in lines 6–32. To describe a process state, we introduce the following local variables:

```
1  #define V0 0 // likewise, V1 is 1, SE is 2, D0 is 3, D1 is 4
2  symbolic int n /* nr. of correct */, f /* nr. of faulty */, majority; // majority size
3  int nsnt0, nsnt1; // counters for 0s and 1s sent by the correct processes
4  assume(n > 1 && n > 3 * f); // the resilience condition restricts faults
5  assume(n + 2 == 2 * majority || n + 1 == 2 * majority); // majority = ⌈(n+1)/2⌉
6  active[n − f] proctype Proc() { // run n − f correct processes
7   // control state: initialized with 0 (V0), initialized with 1 (V1),
8   // sent the value (SE), decided on 0 (D0), decided on 1 (D1)
9   byte pc = V0, next_pc = V0;
10  // counters for received 0s and 1s
11  int nrcvd0 = 0, nrcvd1 = 0, next_nrcvd0 = 0, next_nrcvd1 = 0;
12  if :: pc = V0; // non−deterministically initialize with 0 or 1
13     :: pc = V1; fi;
14  do :: atomic { // a single indivisible step
15    havoc(next_nrcvd0); havoc(next_nrcvd1); // forget variable values
16    // update message counters (up to f messages from the Byzantine processes)
17    assume(nrcvd0 <= next_nrcvd0 && next_nrcvd0 <= nsnt0 + f);
18    assume(nrcvd1 <= next_nrcvd1 && next_nrcvd1 <= nsnt1 + f);
19    // compute the new state and send messages, if needed
20    if :: pc == V0 −> next_pc = SE; nsnt0++; // send 0
21       :: pc == V1 −> next_pc = SE; nsnt1++; // send 1
22       :: pc == SE && next_nrcvd0 >= majority −> next_pc = D0; // decide on 0
23       :: pc == SE && next_nrcvd1 >= majority −> next_pc = D1; // decide on 1
24       :: pc == SE && next_nrcvd0 < majority && next_nrcvd1 < majority
25          −> next_pc = SE; // wait for more messages
26       :: pc == D0 || pc == D1 −> next_pc = pc; // self−loop
27    fi;
28    // update local variables
29    pc = next_pc; nrcvd0 = next_nrcvd0; nrcvd1 = next_nrcvd1;
30    next_pc = 0; next_nrcvd0 = 0; next_nrcvd1 = 0;
31  } od; // next step
32 }
33 // atomic propositions
34 atomic ex_D0 = some(Proc:pc == D0); atomic ex_D1 = some(Proc:pc == D1);
35 atomic all_decide = all(Proc:pc == D0 || Proc:pc == D1);
36 atomic ex_V0 = some(Proc:pc == V0); atomic ex_V1 = some(Proc:pc == V1);
37 atomic in_transit0 = some(Proc:nrcvd0 < nsnt0);
38 atomic in_transit1 = some(Proc:nrcvd1 < nsnt1);
39 // LTL formulae
40 ltl agreement { [](!ex_D0 || !ex_D1) }
41 ltl termination { (<>[](!in_transit0 && !in_transit1)) −> <>all_decide }
42 ltl validity0 { <>(ex_D0) −> ex_V0 }
43 ltl validity1 { <>(ex_D1) −> ex_V1 }
```

Fig. 3. Modeling Naïve Voting in Parametric Promela

– pc to store the algorithm's control state, that is, whether a process is initialized with values 0 and 1 (i.e., pc=V0 and pc=V1 resp.), sent a message (pc=SE), decided on values 0 and 1 (i.e., pc=D0 and pc=D1 resp.)
– nrcvd0 and nrcvd1 to store the number of zeroes and ones received from the correct *and* Byzantine processes; and
– next-state variables next_pc, next_nrcvd0, and next_nrcvd1 that are used to perform a process step.

An initial process state is chosen non-deterministically in lines 12–13.

A single process step is encoded as an atomic block in lines 14–31, which corresponds to an indivisible receive-compute-send step. In lines 15–18, a process possibly receives new messages: by invoking havoc(x), we forget the contents of a variable x, and by writing assume(e), we restrict the variable values to those that satisfy a logical expression e. Note that the statements havoc and assume do not belong to the standard Promela; they belong to parametric Promela and are inspired by the similar statements in Boogie [2]. Lines 20–27 encode the computations that can be found in pseudo-code in Figure 1. Like in Promela, a process non-deterministically picks an option of the form "`:: guard -> actions`", if `guard` evaluates to true, and executes `actions`.

To specify temporal properties, we first define atomic propositions in lines 34–38. The keywords some and all correspond to existential and universal quantification over the processes; they belong to parametric Promela. In lines 40–43, define LTL formulas that capture the properties of consensus (cf. Section 2).

Promela code in Figure 3 models the informal pseudo code of Naïve Voting. Note that the manual translation from pseudo code is straightforward, except for one thing: It may seem more honest to maintain sets of sent and received messages, instead of storing only integer message counters such as nrcvd0 and nsnt0. It has been proven that modeling with sets is equivalent (bisimilar) to modeling with message counters [14]. Obviously, modeling with message counters produces smaller transition systems (cf. [9]).

3.2 Threshold Automata

Our code in parametric Promela has several features: (i) each atomic step is encoded as an imperative sequence of statements, (ii) and the processes explicitly store the number of received messages in local variables such as nrcvd0 and nrcvd1. One can argue that this level of detail is not necessary, and it makes the verification problem harder. Threshold automata are a more abstract model for threshold-guarded fault-tolerant distributed algorithms [11], as they enable guarded transitions as soon as sufficiently many messages have been sent. Intuitively, the reception variables nrcvd0 and nrcvd1 are bypassed by such modeling. In this section, we introduce threshold automata in a way that explains how automata capture local transitions of individual processes. The semantics of threshold automata are then defined via counter systems in Section 4 that model runs of collections of processes, that is, distributed computations.

We model Naïve Voting with the threshold automaton shown in Figure 2. Its code in the `.ta` input format of ByMC is shown in Figure 4. We are running

```
thresholdAutomaton Proc {
  local pc; /* control locations:
               in V0 and V1, initialized with 0 and 1 resp.,
               in D0 and D1, decided on 0 and 1 resp., in SE, sent the initial value */
  shared nsnt0, nsnt1; /* the number of 0s and 1s sent by the correct processes */
  parameters N, T, F; /* parameter variables */
  assumptions (0) { N > 3 * T; T >= F; T >= 1; } /* resilience condition */
  locations (0) { locV0: [0]; locV1: [1]; locSE: [2]; locD0: [3]; locD1: [4];}// local states
  inits (0) { /* initial constraints */
    (locV0 + locV1) == N - F; locSE == 0; locD0 == 0; locD1 == 0;
    nsnt0 == 0; nsnt1 == 0;
  }
  rules (0) { /* a set of rules */
    /* send message 0 (resp. 1) when initialized with value 1 (resp. 1) */
    0: locV0 -> locSE when (true) do { nsnt0' == nsnt0 + 1; nsnt1' == nsnt1; };
    1: locV1 -> locSE when (true) do { nsnt0' == nsnt0; nsnt1' == nsnt1 + 1; };
    2: locSE -> locD0 /* decide on value 0 */
         when (2 * (nsnt0 + F) >= N + 1) do { unchanged(nsnt0, nsnt1); };
    3: locSE -> locD1 /* decide on value 1 */
         when (2 * (nsnt1 + F) >= N + 1) do { unchanged(nsnt0, nsnt1); };
    /* self loops */
    4: locSE -> locSE when (true) do { unchanged(nsnt0, nsnt1); };
    5: locD0 -> locD0 when (true) do { unchanged(nsnt0, nsnt1); };
    6: locD1 -> locD1 when (true) do { unchanged(nsnt0, nsnt1); };
  }
  specifications (0) { /* LTL formulas */
   agreement: [](locD0 == 0 || locD1 == 0);
   validity0: <>(locD0 != 0) -> locV0 != 0;
   validity1: <>(locD1 != 0) -> locV1 != 0;
   termination:
    <>[](locV0 == 0 && locV1 == 0 && (2 * nsnt0 < N + 1 || locSE == 0)
                                  && (2 * nsnt1 < N + 1 || locSE == 0))
    -> <>(locD0 != 0 || locD1 != 0);
  }
} /* Proc */
```

Fig. 4. A threshold automaton for Naïve Voting in the `.ta` format

$n - f$ instances of the threshold automaton; each instance is modelling a correct process. The automata operate on shared variables such as nsnt_0 and nsnt_1, which can be only incremented. A threshold automaton resides in a *local state* from a finite set $\mathcal{L}$, e.g., in our example, $\mathcal{L} = \{\mathsf{V0}, \mathsf{V1}, \mathsf{SE}, \mathsf{D0}, \mathsf{D1}\}$. A *rule* (corresponding to an edge in Figure 2) can move an automaton from one local state to another, provided that the shared variables in the current global state satisfy the rule's threshold guard, e.g., $2{\cdot}(\mathsf{nsnt}_0{+}f) \geq n{+}1$. If a rule is labeled with an increment of a shared variable, e.g., $\mathsf{nsnt}_0\mathsf{++}$, then the shared variable is updated accordingly.

4 Theoretical Background

4.1 System

We assume fixed three finite sets: the set $\mathcal{L}$ contains the *local states*, the set Γ contains the *shared variables* that range over non-negative integers, and the set Π contains the *parameters* that range over non-negative integers.

Configurations Σ and I. A configuration is a vector $\sigma = (\boldsymbol{\kappa}, \mathbf{g}, \mathbf{p})$, where $\sigma.\boldsymbol{\kappa}$ is a vector of *counter values*, $\sigma.\mathbf{g}$ is a vector of *shared variable values*, and $\sigma.\mathbf{p} = \mathbf{p}$ is a vector of *parameter values*. In $\sigma.\boldsymbol{\kappa}$ we store for each local state ℓ, how many processes are in this state. All values are non-negative integers. In every initial configuration global variables have value zero, and all "modelled" processes are in initial locations. If specifications do not limit the behavior of faulty processes (which is typically the case with Byzantine faults), we only model the correct processes explicitly, while the impact of faulty processes is modelled as non-determinism in the environment.

Threshold Guards are defined according to the following grammar:

$$
\begin{aligned}
&\mathit{Guard} ::= \mathit{Int} \cdot \mathit{Shared} \geq \mathit{LinForm} \mid \mathit{Int} \cdot \mathit{Shared} < \mathit{LinForm}\\
&\mathit{LinForm} ::= \mathit{Int} \mid \mathit{Int} \cdot \mathit{Param} \mid \mathit{Int} \cdot \mathit{Param} + \mathit{LinForm}\\
&\mathit{Shared} ::= \langle \text{a variable from } \Gamma \rangle\\
&\mathit{Param} ::= \langle \text{a variable from } \Pi \rangle\\
&\mathit{Int} ::= \langle \text{an integer} \rangle
\end{aligned}
$$

Transition relation R. A *transition* is a pair $t = (\mathit{rule}, \mathit{factor})$ of a rule of the TA and a non-negative integer called the *acceleration factor*, or just factor for short. If the factor is always 1, this corresponds that at each step exactly one processes takes a step, that is, interleaving semantics. Having factors greater than 1 permits a specific form of acceleration where an arbitrary number of processes that are ready to execute a rule can do that at the same time.

Transition t is *applicable (or enabled)* in configuration σ, if the guard of $t.\mathit{rule}$ evaluates to true, and $\sigma.\boldsymbol{\kappa}[t.\mathit{from}] \geq t.\mathit{factor}$. Configuration σ' is the result of applying the enabled transition t to σ, and write $\sigma' = t(\sigma)$, if

- $\sigma'.\mathbf{g} = \sigma.\mathbf{g} + t.\mathit{factor} \cdot t.\mathbf{u}$ and $\sigma'.\mathbf{p} = \sigma.\mathbf{p}$
- if $t.\mathit{from} \neq t.\mathit{to}$ then

$$
\begin{aligned}
\psi &::= \mathit{pform} \mid \mathbf{G}\,\psi \mid \mathbf{F}\,\psi \mid \psi \wedge \psi \\
\mathit{pform} &::= \mathit{cform} \mid \mathit{gform} \vee \mathit{cform} \\
\mathit{cform} &::= \bigvee_{\ell \in Locs} \boldsymbol{\kappa}[\ell] \neq 0 \mid \bigwedge_{\ell \in Locs} \boldsymbol{\kappa}[\ell] = 0 \mid \mathit{cform} \wedge \mathit{cform} \\
\mathit{gform} &::= \mathit{guard} \mid \neg \mathit{gform} \mid \mathit{gform} \wedge \mathit{gform}
\end{aligned}
$$

Table 1. The syntax of $\mathsf{ELTL_{FT}}$-formulas [10]: *pform* defines propositional formulas, and ψ defines temporal formulas. We assume that $Locs \subseteq \mathcal{L}$ and $\mathit{guard} \in \Phi^{\mathrm{rise}} \cup \Phi^{\mathrm{fall}}$.

  - $\sigma'.\boldsymbol{\kappa}[t.\mathit{from}] = \sigma.\boldsymbol{\kappa}[t.\mathit{from}] - t.\mathit{factor}$,
  - $\sigma'.\boldsymbol{\kappa}[t.\mathit{to}] = \sigma.\boldsymbol{\kappa}[t.\mathit{to}] + t.\mathit{factor}$, and
  - $\forall \ell \in \mathcal{L} \setminus \{t.\mathit{from}, t.\mathit{to}\}$ it holds that $\sigma'.\boldsymbol{\kappa}[\ell] = \sigma.\boldsymbol{\kappa}[\ell]$
- if $t.\mathit{from} = t.\mathit{to}$ then $\sigma'.\boldsymbol{\kappa} = \sigma.\boldsymbol{\kappa}$

Finally, the transition relation $R \subseteq \Sigma \times \Sigma$ of the counter system is defined as follows: $(\sigma, \sigma') \in R$ iff there is a rule $r \in \mathcal{R}$ and a factor $k \in \mathbb{N}_0$ such that $\sigma' = t(\sigma)$ for $t = (r, k)$.

Observe that configurations, transitions, guard, etc. can be encoded in linear integer arithmetic.

4.2 Safety and liveness specifications

Using counter systems, we can also easily express the temporal properties, e.g., those of Naïve Voting. To this end, for every local state $\ell \in \mathcal{L}$, we introduce a proposition "$\boldsymbol{\kappa}_\ell = 0$", which tests that there are no processes in ℓ. Since threshold automata do not explicitly track received messages, the assumption of reliable communication is modeled as a fairness assumption over local states and actions. The following formula captures the required fairness, that is, (i) eventually all processes leave their initial state $\mathsf{V0}$ or $\mathsf{V1}$, and (ii) if threshold guards become true, then eventually all processes fire the corresponding rules and thus evacuate the local state SE (the latter implication is written as disjunction):

$$
\begin{aligned}
&\mathbf{F}\,\mathbf{G}\,\big(\boldsymbol{\kappa}_{\mathsf{V0}} = 0 \ \wedge \ \boldsymbol{\kappa}_{\mathsf{V1}} = 0 \\
&\quad \wedge \ (2 \cdot \mathsf{nsnt}_0 < n + 1 \ \vee \ \boldsymbol{\kappa}_{\mathsf{SE}} = 0) \ \wedge \ (2 \cdot \mathsf{nsnt}_1 < n + 1 \ \vee \ \boldsymbol{\kappa}_{\mathsf{SE}} = 0)\big) \quad \text{(RC)}
\end{aligned}
$$

Agreement (A), Validity (V), and Termination (T) can be written as follows:

$$\mathbf{G}\,(\boldsymbol{\kappa}_{\mathsf{D0}} = 0 \vee \boldsymbol{\kappa}_{\mathsf{D1}} = 0) \tag{A}$$

$$\mathbf{F}\,(\boldsymbol{\kappa}_{\mathsf{D0}} \neq 0) \rightarrow \boldsymbol{\kappa}_{\mathsf{V0}} \neq 0 \wedge \mathbf{F}\,(\boldsymbol{\kappa}_{\mathsf{D1}} \neq 0) \rightarrow \boldsymbol{\kappa}_{\mathsf{V1}} \neq 0 \tag{V}$$

$$RC \rightarrow \mathbf{F}\,(\boldsymbol{\kappa}_{\mathsf{V0}} = 0 \wedge \boldsymbol{\kappa}_{\mathsf{V1}} = 0 \wedge \boldsymbol{\kappa}_{\mathsf{SE}} = 0) \tag{T}$$

In [12], we have introduced a bounded model checking technique with SMT that checks reachability in counter systems of threshold automata for all combinations of the parameters. We proved that if a configuration is reachable, then

there is a short schedule that reaches this configuration. As a result, bounded model checking is a complete method for reachability checking in our case. In [10], this technique was extended to $\mathsf{ELTL_{FT}}$ — a fragment of $\mathsf{ELTL}(\mathsf{F},\mathsf{G})$, which allows us to verify safety and liveness of counter systems of threshold automata. The syntax of $\mathsf{ELTL_{FT}}$ is given in Table 1. We use this logic to express *counterexamples*, that is, negations of the safety and liveness specifications from above.

For instance, the negation of agreement and termination in Equations (A) and (T) fit into $\mathsf{ELTL_{FT}}$, and can be written as follows:

$$\mathbf{F}\left(\boldsymbol{\kappa}_{\mathsf{D0}} \neq 0 \wedge \boldsymbol{\kappa}_{\mathsf{D1}} \neq 0\right) \tag{NA}$$

$$RC \wedge \mathbf{G}\left(\boldsymbol{\kappa}_{\mathsf{V0}} \neq 0 \vee \boldsymbol{\kappa}_{\mathsf{V1}} \neq 0 \vee \boldsymbol{\kappa}_{\mathsf{SE}} \neq 0\right) \tag{NT}$$

Technically, the negation of the formula for validity given in Equation (V) does not belong to the fragment $\mathsf{ELTL_{FT}}$. However, it can be easily rewritten as two formulas, for the values of i equal to 0 and 1:

$$\mathbf{F}\left(\boldsymbol{\kappa}_{\mathsf{D}_i} \neq 0\right) \wedge \boldsymbol{\kappa}_{\mathsf{V}_i} = 0 \tag{NV$_\mathrm{i}$}$$

5 Parameterized model checking by schema enumeration

Our verification technique consists of the following steps: From the $\mathsf{ELTL_{FT}}$ specifications, our tool enumerates all shapes counterexamples can have. Each of these shapes is encoded as an SMT query, and using SMT solvers, our tool checks for each shape, whether there exists a run of the system that has this shape. Such a run would then be a witness to the violation of a specification.

Consider the agreement property (A) of Naïve Voting. A counterexample is a run of the system that starts in an initial state and satisfies its negation:

$$\mathbf{F}\left(\boldsymbol{\kappa}_{\mathsf{D0}} \neq 0 \wedge \boldsymbol{\kappa}_{\mathsf{D1}} \neq 0\right)$$

Each counterexample thus (i) satisfies the constraints for initial states, and (ii) is a sequence of applicable transitions, that (iii) end up in a state where $(\boldsymbol{\kappa}_{\mathsf{D0}} \neq 0 \wedge \boldsymbol{\kappa}_{\mathsf{D1}} \neq 0)$ holds. Indeed checking (A) boils down to checking reachability of a state that satisfies $(\boldsymbol{\kappa}_{\mathsf{D0}} \neq 0 \wedge \boldsymbol{\kappa}_{\mathsf{D1}} \neq 0)$. Our technique from [12] enumerates all shapes of such counterexamples.

The central notion is a *simple schema*:

$$\{pre\}r_1^*, \ldots, r_k^*\{post\}$$

where $pre, post \subseteq$ are contraints that encode evaluation of guards, and constraints on the counters (e.g., $\boldsymbol{\kappa}_{\mathsf{D0}} \neq 0$). Thus, the schema captures that *pre* holds, then some transitions with rules $r_1^*, \ldots, r_k^*$ are ececuted to reach a state where *post* holds. We denote a simple schema by S. A schema is then a concatenation of simple schemas $S_1, S_2, \ldots S_k$, for some k.

For our example, the technique from [12] would generate among others, a schema like the following

$$
\begin{aligned}
S_1, S_2, S_3 = \\
&\{\boldsymbol{\kappa}_{\mathsf{V0}} + \boldsymbol{\kappa}_{\mathsf{V1}} = n\} r_1^*, \ldots, r_4^* \\
&\{2 \cdot (\mathsf{nsnt}_0 + f) \geq n + 1\} r_1^*, \ldots, r_4^* \\
&\{(2 \cdot (\mathsf{nsnt}_0 + f) \geq n + 1), (2 \cdot (\mathsf{nsnt}_1 + f) \geq n + 1)\} r_1^*, \ldots, r_4^* \\
&\{(2 \cdot (\mathsf{nsnt}_0 + f) \geq n + 1), (2 \cdot (\mathsf{nsnt}_1 + f) \geq n + 1), (\boldsymbol{\kappa}_{\mathsf{D0}} \neq 0 \wedge \boldsymbol{\kappa}_{\mathsf{D1}} \neq 0)\}
\end{aligned}
$$

that is, initially, all of the n processes are in the initial locations $\mathsf{V0}$ and $\mathsf{V1}$, then after application of some rules one of the threshold guards becomes true, then after another application of some rules both guards are true and finally a bad state is reached. The SMT solver now has to find whether an executions exists that has that form. This is done by replacing each Kleene star by a distinct variable that encodes how often a rule r is applied.

A different schema can be obtained by changing the order in which the two threshold guards become true. In general each possible order generates a different schema. The number of different schemas to be checked is factorial in the number of guards [12]. As our benchmarks have only a small number of guards, the number of calls to the SMT solver is still practical.

5.1 Checking a single lasso schema with SMT

In [10] we prove that for our counter systems, a counterexample to a liveness specification has lasso shape, that is:

$$S_1 \ldots S_k (S_{k+1} \ldots S_{k+m})^\omega$$

In this way we obtain a finite representation of an infinite execution, which again can be checked with an SMT solver.

Thus, our tool generates multiple schemas: for each safety or liveness specification, a different schema is obtained by changing the order in which the threshold guards become true. A detailed algorithm for constructing schemas is presented in [10][Fig. 10]. In a nutshell, the algorithm constructs a graph that represents the partial order on when propositions and threshold guards evaluate to true in an execution, e.g., the one in Figure 5. Each linear extension of this partial order then defines a sequence on which propositions and guards become true. Two neighboring elements in the sequence are the *pre* and *post* of a simple schema; the concatenation of all these simple schemas is the schema our tool checks for satisfiability.

Our tool encodes each schema in SMT and then calls a back-end solver in order to check whether the schema generates a counterexample. In [10], we explained the SMT encoding. As the schemas are independent, these checks can be done in parallel. We have implemented and exploited this feature in [15]. As [15] was concerned with synthesis, we did not discuss the effects of parallelization there. In the following we discuss and compare the sequential and the parallel approaches.

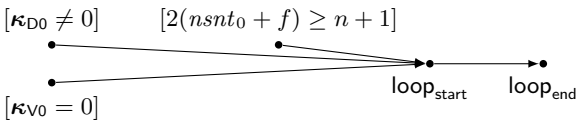

Fig. 5. The graph constructed from the automata guards and formula (NV_0)

Sequential schema enumeration. In the sequential mode, the schemas are simply checked one-by-one until either a counterexample is found, or all schemas have been enumerated and no counterexample has been found. (Detailed pseudo-code of the function `check_one_order` can be found in Figure 10 of [10].)

```
1   for each linear order ⪯ of graph 𝒢 {
2     if check_one_order(TA, φ, 𝒢, ⪯) = witness(SMT_model)
3       report SMT_model as a counterexample
4   }
5   report specification holds
```

Parallel schema enumeration. In the MPI mode, the tool runs as a system of N processes, one per CPU; the physical arrangement of the CPUs depends on the cluster configuration. Every process is assigned a unique value *rank* from 0 to $N-1$: The process with $rank = 0$ is the master, whereas the other processes are the workers. Every process is enumerating the schemas as in the sequential mode but checks a schema only if the schema's sequence number i matches the rule: $(i \mod N) = rank$. In order to terminate quickly when one process has found a bug, the workers asynchronously communicate with the master. After leaving the loop, the workers communicate with the master to deliver a counterexample, if one was found. For presentation, we assume that the master can send to and receive messages from itself.

```
1   i := 0; found := false
2   for each linear order ⪯ of graph 𝒢 {
3     if rank = i and check_one_order(TA, φ, 𝒢, ⪯) = witness(SMT_model)
4       found := true
5       send BUG to master // notify the master
6     if received BUG from any
7       if rank = master { send BUG to all } // notify the workers
8       break
9     i := i + 1
10  }
11  results = gather found master // the workers send their 'found' flags to the master
12  if rank = master {
13    if ∃w : results[w] = true
14      send WITNESS<w> to all // pick one counterexample and declare it a witness
15      if w = master report SMT_model as a counterexample
16      else { receive CEX<model> from w; report model as a counterexample }
17    else { send WITNESS<⊥> to all; report specification holds }
```

```
18    } else {
19      receive WITNESS<w> from master
20      if w = rank { send CEX<SMT_model> to master } // I am the witness
21    } // finish and clean up stale MPI messages on exit
```

6 Benchmarks and Experiments

Byzantine model checker is written in OCaml. Its source code and the virtual machines are available from the tool web page [4]. For the experiments conducted in this paper, we used Z3 4.6.0 [4] as a back-end SMT solver, which was linked to ByMC via Z3 OCaml bindings.

In earlier work [9], we encoded our benchmarks in Parametric Promela, using a shared variable to record the number of processes that have sent a message, and using for each process a local variable that records how many messages a process received. For this modeling we presented a data abstraction and counter abstraction in [8]. To compare later verification techniques with these initial results, we kept that encoding, although the newer techniques rest on a more abstract model of threshold automata, which have finitely many local states.

The threshold automata constructed by data abstraction are significantly larger than threshold automata constructed by a human expert. To see the influence of these modeling decisions on the verification results, we manually encoded our benchmarks as threshold automata. These benchmarks are available from our benchmark repository [5]. Table 2 compares the size of the threshold automata that are: (1) produced automatically by abstraction and (2) hand-coded. The essential features of the automata are: the number of local states $|\mathcal{L}|$, the number of rules $|\mathcal{R}|$, and the numbers of the guards $|\Phi^{\text{rise}}|$ and $|\Phi^{\text{fall}}|$, that is, the guards of the form $x \geq \ldots$ and $x < \ldots$ respectively. Moreover, due to data abstraction, we had to consider several cases that differ in the order between the thresholds. They are mentioned in the column "Case".

Table 2 shows the verification results for benchmarks in Promela as well as threshold automata. We ran the sequential schema enumeration (SEQ, [10]) and the parallel schema checking technique (MPI) that is presented in this paper. The parallel experiments were run at Vienna Scientific Cluster using 256 CPU cores. For each benchmark, we picked the most challenging specifications — many of them are liveness properties — and show experimental results for them. (Needless to say, we did not run the MPI technique on the benchmarks that could be enumerated with the sequential technique in seconds.) Two columns show the essential features of the enumerated schemas: "number" displays the total number of explored schemas, and "length avg" displays the average length of schemas. For both techniques, we report the computation times and maximal memory usage during a run. For the MPI experiments, we report the average time per CPU core (column "MPI avg") as well as the maximum time per CPU core (column "MPI max"). The deviation from the average case is negligible.

[4] `http://forsyte.at/software/bymc`

[5] `https://github.com/konnov/fault-tolerant-benchmarks/tree/master/isola18`

Table 2. The experiments with the sequential (SEQ) and parallel (MPI) techniques on two kinds of inputs: Promela (white rows) and threshold automata (gray rows). The sequential experiments were run with GNU parallel [23] at AMD Opteron® 6272, 32 cores, 192 GB. The MPI benchmarks were run at Vienna Scientific Cluster 3 using 16 nodes × 16 cores (256 processes). The symbol "◷" indicates timeout of 24 hours.

#	Input	Case	Threshold Automaton				Schemas		Time, seconds			Mem, GB	
								length	SEQ	MPI		SEQ	MPI
	FTDA	(if more than one)	$\lvert\mathcal{L}\rvert$	$\lvert\mathcal{R}\rvert$	$\lvert\Phi^{\text{rise}}\rvert$	$\lvert\Phi^{\text{fall}}\rvert$	number	avg		avg	max		avg
1	**frb**	-	7	14	1	0	5	34	1	-	-	0.1	-
2	**frb**	*hand-coded TA*	4	9	1	1	70	38	1	-	-	0.1	-
3	**strb**	-	7	21	3	0	18	72	1	-	-	0.1	-
4	**strb**	*hand-coded TA*	4	8	2	0	38	22	1	-	-	0.1	-
5	**nbacg**	-	24	64	4	0	90	243	6	-	-	0.1	-
6	**nbacg**	*hand-coded TA*	8	16	0	1	5	54	1	-	-	0.1	-
7	**nbacr**	-	77	1031	6	0	517	2489	523	-	-	0.7	-
8	**nbacr**	*hand-coded TA*	7	16	0	1	18	63	1	-	-	0.1	-
9	**aba**	$\frac{n+t}{2} = 2t+1$	37	202	6	0	1172	850	659	12	13	1.0	0.2
10	**aba**	$\frac{n+t}{2} > 2t+1$	61	425	8	0	5204	2112	53992	1440	1442	7.2	0.6
11	**aba**	*hand-coded TA*	5	10	2	2	542	57	14	-	-	0.1	-
12	**cbc**	$\lfloor\frac{n}{2}\rfloor < n-t \wedge f = 0$	164	2064	0	0	2	8168	1603	290	290	9.3	0.2
13	**cbc**	$\lfloor\frac{n}{2}\rfloor = n-t \wedge f = 0$	73	470	0	0	2	1790	27	9	9	0.6	0.1
14	**cbc**	$\lfloor\frac{n}{2}\rfloor < n-t \wedge f > 0$	165	2072	0	1	4	10213	10024	4943	4943	18.8	0.5
15	**cbc**	$\lfloor\frac{n}{2}\rfloor = n-t \wedge f > 0$	74	476	0	1	4	2258	273	47	47	1.5	0.1
16	**cbc**	*hand-coded TA*	7	14	0	1	5	56	1	-	-	0.1	-
17	**cf1s**	$f = 0$	41	280	4	0	90	770	45	5	8	0.2	0.1
18	**cf1s**	$f = 1$	41	280	4	1	523	787	257	6	6	0.4	0.1
19	**cf1s**	$f > 1$	68	696	6	1	3429	2132	10346	29	29	3.8	0.2
20	**cf1s**	*hand-coded TA*	9	26	3	3	13700	122	687	6	8	2.1	0.1
21	**c1cs**	$f = 0$	101	1285	8	0	251	460	331	38	38	0.8	0.1
22	**c1cs**	$f = 1$	70	650	6	1	448	303	239	11	11	0.4	0.1
23	**c1cs**	$f > 1$	101	1333	8	1	2100	404	1865	89	89	1.3	0.4
24	**c1cs**	*hand-coded TA*	9	30	7	3	$3.2 \cdot 10^6$	≈ 400	◷	979	981	17.3	1.6
25	**bosco**	$\lfloor\frac{n+3t}{2}\rfloor + 1 = n-t$	28	152	6	0	20	423	4	3	4	0.1	0.1
26	**bosco**	$\lfloor\frac{n+3t}{2}\rfloor + 1 > n-t$	40	242	8	0	70	1038	29	6	6	0.2	0.1
27	**bosco**	$\lfloor\frac{n+3t}{2}\rfloor + 1 < n-t$	32	188	6	0	20	476	4	4	4	0.1	0.1
28	**bosco**	$n > 5t \wedge f = 0$	82	1372	12	0	3431	27	265	35	35	0.3	0.4
29	**bosco**	$n > 7t$	90	1744	12	0	3431	179	1325	52	52	1.0	0.6
30	**bosco**	*hand-coded TA*	8	20	3	4	3429	43	82	4	4	0.2	0.1

As expected, the hand-coded benchmarks are usually verified much faster. Interestingly, the manually constructed threshold automaton for one-step consensus (c1cs [3]) has more threshold guards than the abstract one: We had to more accurately encode algorithm's decisions, crash faults, and fairness. The sequential technique times out on this benchmark. The parallel technique takes about seven times longer than with the automatic abstraction.

The parallel technique benefits from running on multiple cores, though the actual gains from parallelism depend on the benchmark. As in our experiments the verification times of a single schema negligibly deviate from the average case, the uniform distribution of schemas among the nodes seems sufficient. However, one can construct threshold automata that produce schemas whose verification times significantly vary from each other. We conjecture that an implementation with a dynamic balancer would make better use of cluster resources.

7 Conclusions

We presented our tool ByMC, and compared its sequential verification implementation to its parallel one. Moreover, by experimental evaluation we showed that manual abstractions give us threshold automata that can be verified significantly faster than those that result from automatic abstraction.

We observe that the sizes of the manually constructed threshold automata are not significantly larger than the (manually crafted) models of round-based distributed consensus presented in [17]. In their theory, threshold-guarded expressions also play a central role. Our gains in efficiency in this paper—due to manual encodings—show that the discrepancy was a result of automatic abstraction and not of the technique that uses threshold automata as its input.

We needed from one to three hours per benchmark to specify and debug a threshold automaton, while it usually took us less than 30 minutes to specify the same benchmark in Parametric Promela. The most difficult part of the encoding with threshold automata was to faithfully express fairness constraints over shared variables and process counters. In case of Parametric Promela, fairness constraints were much easier to write, as one could refer to the shared and local variables, which count the number of sent and received messages respectively.

Acknowledgments. We are grateful to our past and present collaborators Annu Gmeiner, Marijana Lazić, Ulrich Schmid, and Helmut Veith, who contributed to many of the described ideas that are now implemented in ByMC.

References

1. Attiya, H., Welch, J.: Distributed Computing. Wiley, 2nd edn. (2004)
2. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: Formal Methods for Components and Objects. pp. 364–387 (2005)

3. Brasileiro, F.V., Greve, F., Mostéfaoui, A., Raynal, M.: Consensus in one communication step. In: PaCT. LNCS, vol. 2127, pp. 42–50 (2001)
4. 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)
5. Dobre, D., Suri, N.: One-step consensus with zero-degradation. In: DSN. pp. 137–146 (2006)
6. Guerraoui, R.: Non-blocking atomic commit in asynchronous distributed systems with failure detectors. Distributed Computing 15(1), 17–25 (2002)
7. Holzmann, G.: The SPIN Model Checker. Addison-Wesley (2003)
8. 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)
9. John, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Towards modeling and model checking fault-tolerant distributed algorithms. In: SPIN. LNCS, vol. 7976, pp. 209–226 (2013)
10. 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)
11. 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)
12. 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)
13. 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)
14. Konnov, I.V., Widder, J., Spegni, F., Spalazzi, L.: Accuracy of message counting abstraction in fault-tolerant distributed algorithms. In: VMCAI. pp. 347–366 (2017)
15. Lazic, 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)
16. Lynch, N.: Distributed Algorithms. Morgan Kaufman (1996)
17. Marić, O., Sprenger, C., Basin, D.A.: Cutoff Bounds for Consensus Algorithms. In: CAV. pp. 217–237 (2017)
18. Mostéfaoui, A., Mourgaya, E., Parvédy, P.R., Raynal, M.: Evaluating the condition-based approach to solve consensus. In: DSN. pp. 541–550 (2003)
19. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. J.ACM 27(2), 228–234 (1980)
20. Raynal, M.: A case study of agreement problems in distributed systems: Non-blocking atomic commitment. In: HASE. pp. 209–214 (1997)
21. Song, Y.J., van Renesse, R.: Bosco: One-step Byzantine asynchronous consensus. In: DISC. LNCS, vol. 5218, pp. 438–450 (2008)
22. Srikanth, T., Toueg, S.: Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. Dist. Comp. 2, 80–94 (1987)
23. Tange, O., et al.: Gnu parallel-the command-line power tool. The USENIX Magazine 36(1), 42–47 (2011)
24. Tseng, L.: Voting in the presence of byzantine faults. In: Dependable Computing (PRDC). pp. 1–10. IEEE (2017)