HAL
archives-ouvertes.fr

Verifying Safety of Synchronous Fault-Tolerant Algorithms by Bounded Model Checking

Ilina Stoilkovska, Igor Konnov, Josef Widder, Florian Zuleger

► **To cite this version:**

Ilina Stoilkovska, Igor Konnov, Josef Widder, Florian Zuleger. Verifying Safety of Synchronous Fault-Tolerant Algorithms by Bounded Model Checking. 2018. <hal-01925653>

HAL Id: hal-01925653

https://hal.inria.fr/hal-01925653

Submitted on 16 Nov 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Verifying Safety of Synchronous Fault-Tolerant Algorithms by Bounded Model Checking

Ilina Stoilkovska[1], Igor Konnov[2], Josef Widder[1], and Florian Zuleger[1]

[1] TU Wien
[2] University of Lorraine, CNRS, Inria, LORIA

Abstract. Owing to well-known limitations of what can be achieved in purely asynchronous systems, many fault-tolerant distributed algorithms are designed for synchronous or round-based semantics. In this paper, we introduce the synchronous variant of threshold automata, and study their applicability and limitations for the verification of synchronous distributed algorithms.
We show that in general, the reachability problem is undecidable for synchronous threshold automata. Still, we show that many synchronous fault-tolerant distributed algorithms have a bounded diameter, although the algorithms are parameterized by the number of processes. Hence, we use bounded model checking for verifying these algorithms.
The existence of bounded diameters is the main conceptual insight in this paper. We compute the diameter of several algorithms and check their safety properties, using SMT queries that 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 4), which makes our approach applicable in practice. For a specific class of algorithms we also establish a theoretical result on the existence of a diameter, providing a first explanation for our experimental results.

1 Introduction

Fault-tolerant distributed algorithms are hard to design and verify. Recently, threshold automata were introduced to model, verify and synthesize *asynchronous* fault-tolerant distributed algorithms [19, 17, 22]. Owing to a well-known impossibility result [16] many distributed computing problems, including consensus, are not solvable in purely asynchronous systems. Thus, synchronous distributed algorithms have been extensively studied [4, 23]. In this paper, we introduce *synchronous* threshold automata, and investigate their applicability and limitations for verification of synchronous fault-tolerant distributed algorithms.

An example of such a synchronous threshold automaton is given in Figure 1; it encodes the synchronous reliable broadcast algorithm from [29]. (The pseudo code is given in Figure 1.) Its semantics is defined in terms of a counter system. For each location $\ell_i \in \{\text{V}0, \text{V}1, \textsc{se}, \textsc{ac}\}$ (a node in the graph), we have a counter κ_i

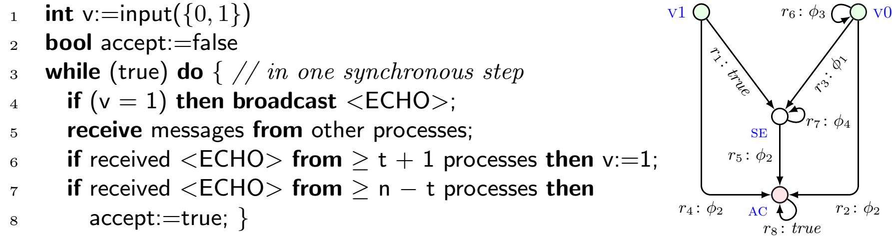

Fig. 1. Pseudo code of synchronous reliable broadcast à la [29], and its STA, with guards: $\phi_1 \equiv \#\{\text{V1}, \text{SE}, \text{AC}\} \geq t+1-f$ and $\phi_2 \equiv \#\{\text{V1}, \text{SE}, \text{AC}\} \geq n-t-f$ and $\phi_3 \equiv \#\{\text{V1}, \text{SE}, \text{AC}\} < t+1$ and $\phi_4 \equiv \#\{\text{V1}, \text{SE}, \text{AC}\} < n-t$.

that stores the number of processes that are in ℓ_i. The 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 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. For example, the guard $\phi_1 \equiv \#\{\text{V1}, \text{SE}, \text{AC}\} \geq t+1-f$ evaluates to true if the number of processes that are either in location V1, SE, or AC is greater than or equal to $t+1-f$.

Synchronous Threshold Automata (STA) model synchronous fault-tolerant distributed algorithms as follows. As the processes send messages based on their current locations, we use the number of processes in given locations to test how many messages of a certain type have been sent. 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 spurious messages in a two-faced manner to some correct processes. Thus, the number of messages received by a process may deviate from the number of correct processes that sent messages. For example, if the guard in line 6 evaluates to true, the $t+1$ received messages may contain up to f messages from faulty processes. In the case that from 1 to t correct processes send `<ECHO>`, faulty processes may "help" some (possibly not all) correct processes to pass over the $t+1$ threshold. In the STA, this is modeled when rules r_3 and r_6 are both enabled. Thus, setting `v:=1` in line 6 is captured by rule r_3 with the guard ϕ_1. Not updating `v` is captured by r_6 with guard ϕ_3. As we capture the effect of the f faulty processes in the guards, we model only the correct processes explicitly, so that a system consists of $n-f$ copies of the STA.

Contributions. We start by introducing synchronous threshold automata (STA) and the counter systems they define.

1. We show that parameterized reachability checking of STA is undecidable.
2. We introduce an SMT-based procedure for finding the diameter of a counter system associated with an STA, i.e., the number of steps in which every con-

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

figuration of the counter system is reachable. By knowing the diameter, we use bounded model checking as a complete verification method [10, 20, 12].

3. For a class of STA that captures several algorithms such as the broadcast algorithm in Figure 1, we prove that a diameter exists. The diameter is a function of the number of guard expressions and the longest path in the automaton, that is, it is independent of the parameters.
4. In our experimental evaluation, our tool verifies the benchmarks by running Z3 [26] and CVC4 [6] as back-end SMT solvers. By tweaking the constraints on the parameters n, t, f, we introduce configurations with more faults than expected, for which our tool automatically finds a counterexample.
5. Our tool verifies several distributed algorithms from the literature: Benchmarks that tolerate Byzantine faults from [29, 9, 8, 7] and benchmarks that tolerate crash faults from [23, 11, 27]. Benchmarks that tolerate send omission faults are from [9, 27]. We are the first to automatically verify the Byzantine and send omission benchmarks. For the crash benchmarks, our new method performs significantly better than the abstraction-based method in [3].

2 Overview of our Approach

Bounded diameter. Consider Figure 1: the processes execute the send, receive, and local computation steps in lock-step. One iteration of the loop is expressed as an STA edge that connects the locations before and after an iteration (i.e., the STA models the loop body of the pseudo code). The location SE encodes that `v` = 1 and `accept` is false. That is, SE is the location in which processes send `<ECHO>` in every round. If a process sets `accept` to true, it goes to location AC. The location where `v` is 1 is encoded by V1, and the where `v` is 0 by V0.

An example execution is depicted in Table 1 on the left. We run $n-f$ copies of the STA in Figure 1. Observe that the guards of the rules r_3 and r_6 are both enabled in the configuration σ_0. One STA uses r_3 to go to SE while the others use the self-loop r_6 to stay in V0. As both rules remain enabled, in every round one more automaton can go to SE. Hence, configuration σ_{t+1} has $t+1$ correct STA in location SE and rule r_6 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 is unbounded for increasing values of the parameter t. (We note that we can obtain longer executions, if some STA use rule r_7). On the right, we see an execution where all STA take r_3 immediately. That is, while configuration σ_{t+3} is reached by a long execution on the left, it is

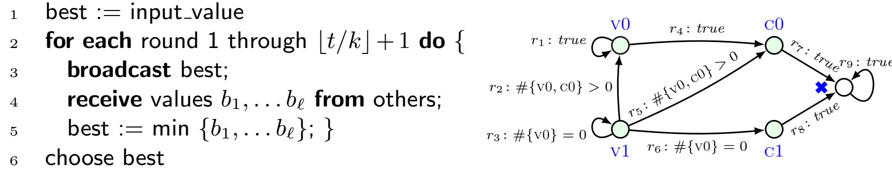

Fig. 2. Pseudo code of *FloodMin* from [11], and STA encoding its loop body, for $k = 1$

reached in just two steps on the right (observe $\sigma'_2 = \sigma_{t+3}$). We are interested if there is a natural number k (which does not depend on the parameters n, f and t) 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*. In Section 5.1 we introduce an SMT-based procedure that enumerates candidates for the diameter bound and checks if the candidate is indeed the diameter; if it finds such a bound, it terminates. For the STA in Figure 1, this procedure computes the diameter 2.

Threshold Automata with traps. In Section 5.2, we define a fragment of threshold automata for which we theoretically guarantee a bounded diameter. For example, the STA in Figure 1 falls in this fragment, and we obtain a guaranteed diameter of ≤ 8. The fragment is defined by two conditions: (i) The STA has a structure that implies monotonicity of the guards: the set of locations that are used in the guards (e.g., $\{\textsc{v1}, \textsc{se}, \textsc{ac}\}$) is closed under the rules, i.e., from each location within the set, the STA can reach only a location in the set. We call guards that have this property *trapped*. (ii) The STA has no cycles, except possibly self-loops.

Bounded Model Checking, Completeness and (Un-)Decidability. The existence of a bounded diameter motivates the use of bounded model checking for verifying safety properties. In Section 6 we give an SMT encoding for checking the violation of a safety property by executions with length up to the diameter. Crucially, this approach is complete because if there is an execution that violates the safety property, there already is a violation by an execution of bounded length. Moreover, we observe that for the STA defined in this paper (with linear guards and linear constraints on the parameters), the SMT encoding results in a formula in Presburger arithmetic (with one quantifier alternation). Hence, checking safety properties (that can be expressed in Presburger arithmetic) is decidable for STA with bounded diameter. We also experimentally demonstrate in Section 7 that current SMT solvers can handle these quantified formulae well. On the contrary, we show in Section 4 that the parameterized reachability problem is undecidable for general STA. This implies that there are STA with unbounded diameter.

Threshold Automata with untrapped guards. The *FloodMin* algorithm in Figure 2 solves the k-set agreement problem. This algorithm is ran by n replicated processes, up to t of which may fail by crashing. For simplicity of presentation,

we consider the case when $k = 1$, which turns k-set agreement into consensus. In Figure 2, on the right, we have the corresponding STA that captures the loop body. The locations C0 and C1 correspond to the case when a process is crashing in the current round and may still manage to send the value 0 and 1 respectively; the process remains in the crashed location "✖" and does not send any messages starting with the next round. We observe that the guard $\#\{\text{V0}, \text{C0}\} > 0$ is not trapped, and our result about trapped guards does not apply. Nevertheless, our SMT-based procedure can again find a diameter of 2. In the same way, we automatically found a bounded diameter for several benchmarks from the literature. It is remarkable that the diameter for the transition relation of the loop body (without the loop condition) is bounded by a constant, independent of the parameters.

Bounded Model Checking of Algorithms with Clean Rounds. The number of loop iterations $\lfloor t/k \rfloor + 1$ of the *FloodMin* algorithm has been designed such that it ensures (together with the environment assumption of at most t crashes) that there is at least one *clean* round in which at most $k - 1$ processes crashed. The correctness of the *FloodMin* algorithm relies on the occurrence of such a clean round. We make use of the existence of clean rounds by employing the following two-step methodology for the verification of safety properties: (i) we find all reachable clean-round configurations, and (ii) check if a bad configuration is reachable from those configurations. Detailed description of this methodology can be found in Section 6. Our method requires the encoding of a clean round as input (e.g., for Figure 2 that no STA are in C0 and C1). We leave detecting and encoding clean rounds automatically from the fault environment for future work.

3 Synchronous Threshold Automata

We introduce the syntax of synchronous threshold automata and give some intuition of the semantics, which we will formalize as counter systems below.

A *synchronous threshold automaton* is the tuple $STA = (\mathcal{L}, \mathcal{I}, \Pi, \mathcal{R}, RC, \chi)$, where $\mathcal{L}$ is a finite set of locations, $\mathcal{I} \subseteq \mathcal{L}$ is a non-empty set of initial locations, Π is a finite set of parameters, $\mathcal{R}$ is a finite set of rules, RC is a resilience condition, and χ is a counter invariant, defined in the following. We assume that the set Π of parameters contains at least the parameter n, denoting the number of processes. We call the vector $\boldsymbol{\pi} = \langle \pi_1, \ldots, \pi_{|\Pi|} \rangle$ the *parameter vector*, and a vector $\mathbf{p} = \langle p_1, \ldots, p_{|\Pi|} \rangle$ is an *instance of* $\boldsymbol{\pi}$, where $\pi_i \in \Pi$ is a parameter, and $p_i \in \mathbb{N}$ is a natural number, for $1 \leq i \leq |\Pi|$, such that $\mathbf{p}[\pi_i] = p_i$ is the value assigned to the parameter π_i in the instance $\mathbf{p}$ of $\boldsymbol{\pi}$. The set of *admissible instances of* $\boldsymbol{\pi}$ is defined as $P_{RC} = \{\mathbf{p} \in \mathbb{N}^{|\Pi|} \mid \mathbf{p} \text{ is an instance of } \boldsymbol{\pi} \text{ and } \mathbf{p} \text{ satisfies } RC\}$. The mapping $N : P_{RC} \to \mathbb{N}$ maps an admissible instance $\mathbf{p} \in P_{RC}$ to the number $N(\mathbf{p})$ of processes that participate in the algorithm, such that $N(\mathbf{p})$ is a linear combination of the parameter values in $\mathbf{p}$.

For example, for the STA in Figure 1, $RC \equiv n > 3t \wedge t \geq f$, hence a vector $\mathbf{p} \in \mathbb{N}^{|\Pi|}$ is an admissible instance of the parameter vector $\boldsymbol{\pi} = \langle n, t, f \rangle$, if

$\mathbf{p}[n] > 3\mathbf{p}[t] \wedge \mathbf{p}[t] \geq \mathbf{p}[f]$. Furthermore, for this STA, $N(\mathbf{p}) = \mathbf{p}[n] - \mathbf{p}[f]$. For the STA in Figure 2, $RC \equiv n > t \wedge t \geq f$, hence the admissible instances satisfy $\mathbf{p}[n] > \mathbf{p}[t] \wedge \mathbf{p}[t] \geq \mathbf{p}[f]$, and we have $N(\mathbf{p}) = \mathbf{p}[n]$.

We introduce *counter atoms* of the form $\psi \equiv \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$, where $L \subseteq \mathcal{L}$ is a set of locations, $\#L$ denotes the total number of processes currently in the locations $\ell \in L$, $\boldsymbol{a} \in \mathbb{Z}^{|\Pi|}$ is a vector of coefficients, $\boldsymbol{\pi}$ is the parameter vector, and $b \in \mathbb{Z}$. We will use the counter atoms for expressing guards and predicates in the verification problem. In the following, we will use two abbreviations: $\#L = \boldsymbol{a} \cdot \boldsymbol{\pi} + b$ for the formula $(\#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b) \wedge \neg(\#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b + 1)$, and $\#L > \boldsymbol{a} \cdot \boldsymbol{\pi} + b$ for the formula $\#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b + 1$.

A *rule* $r \in \mathcal{R}$ is the tuple $(\mathit{from}, \mathit{to}, \varphi)$, where $\mathit{from}, \mathit{to} \in \mathcal{L}$ are locations, and φ is a guard whose truth value determines if the rule r is executed. The guard φ is a Boolean combination of counter atoms. We denote by Ψ the set of counter atoms occurring in the guards of the rules $r \in \mathcal{R}$.

The *counter invariant* χ is a Boolean combination of counter atoms $\#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$, where each atom occurring in χ restricts the number of processes allowed to populate the locations in $L \subseteq \mathcal{L}$.

Counter Systems. The counter atoms are evaluated over tuples $(\kappa, \mathbf{p})$, where $\kappa \in \mathbb{N}^{|\mathcal{L}|}$ is a vector of *counters*, and $\mathbf{p} \in P_{RC}$ is an admissible instance of $\boldsymbol{\pi}$. For a location $\ell \in \mathcal{L}$, the counter $\kappa[\ell]$ denotes the number of processes that are currently in the location ℓ. A counter atom $\psi \equiv \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$ is *satisfied* in the tuple $(\kappa, \mathbf{p})$, that is $(\kappa, \mathbf{p}) \models \psi$, iff $\sum_{\ell \in L} \kappa[\ell] \geq \boldsymbol{a} \cdot \mathbf{p} + b$. The semantics of the Boolean connectives is standard.

A *transition* is a function $t : \mathcal{R} \to \mathbb{N}$ that maps a rule $r \in \mathcal{R}$ to a factor $f = t(r) \in \mathbb{N}$, denoting the number of processes that act upon this rule. Given an instance $\mathbf{p}$ of $\boldsymbol{\pi}$, we denote by $T(\mathbf{p})$ the set $\{t \mid \sum_{r \in \mathcal{R}} t(r) = N(\mathbf{p})\}$ of transitions whose rule factors sum up to $N(\mathbf{p})$.

Given a tuple $(\kappa, \mathbf{p})$ and a transition t, we say that t is *enabled* in $(\kappa, \mathbf{p})$, if

1. for every $r \in \mathcal{R}$, such that $t(r) > 0$, it holds that $(\kappa, \mathbf{p}) \models r.\varphi$, and
2. for every $\ell \in \mathcal{L}$, it holds that $\kappa[\ell] = \sum_{r \in \mathcal{R} \wedge r.\mathit{from} = \ell} t(r)$.

The first condition ensures that processes only use rules whose guards are satisfied, and the second that every process moves in an enabled transition.

Observe that each transition $t \in T(\mathbf{p})$ defines a unique tuple $(\kappa, \mathbf{p})$ in which it is enabled. We call this tuple the *origin* of a transition $t \in T(\mathbf{p})$, and denote it by $o(t)$, such that for every $\ell \in \mathcal{L}$, the following holds: $o(t).\kappa[\ell] = \sum_{r \in \mathcal{R} \wedge r.\mathit{from} = \ell} t(r)$. Similarly, each transition defines a unique configuration that is the result of applying the transition in its origin. We call this configuration the *goal* of the transition $t \in T(\mathbf{p})$, and denote it by $g(t)$, such that for every $\ell \in \mathcal{L}$, we have $g(t).\kappa[\ell] = \sum_{r \in \mathcal{R} \wedge r.\mathit{to} = \ell} t(r)$.

We now define a counter system, for a given $STA = (\mathcal{L}, \mathcal{I}, \Pi, \mathcal{R}, RC, \chi)$, and an admissible instance $\mathbf{p} \in P_{RC}$ of the parameter vector $\boldsymbol{\pi}$.

Definition 1. *A* counter system *w.r.t.* $STA = (\mathcal{L}, \mathcal{I}, \Pi, \mathcal{R}, RC, \chi)$ *and an admissible instance* $\mathbf{p} \in P_{RC}$ *is the tuple* $\mathsf{CS}(STA, \mathbf{p}) = (\Sigma(\mathbf{p}), I(\mathbf{p}), R(\mathbf{p}))$*, where*

- $\Sigma(\mathbf{p}) = \{\sigma = (\kappa, \mathbf{p}) \mid \sum_{\ell \in \mathcal{L}} \sigma.\kappa[\ell] = N(\mathbf{p})$ *and* $\sigma \models \chi\}$ *are the configurations*
- $I(\mathbf{p}) = \{\sigma \in \Sigma(\mathbf{p}) \mid \sum_{\ell \in \mathcal{I}} \sigma.\kappa[\ell] = N(\mathbf{p})\}$ *are the initial configurations*
- $R(\mathbf{p}) \subseteq \Sigma(\mathbf{p}) \times T(\mathbf{p}) \times \Sigma(\mathbf{p})$ *is the transition relation, with* $\langle \sigma, t, \sigma' \rangle \in R(\mathbf{p})$, *if* σ *is the origin and* σ' *the goal of* t. *We write* $\sigma \xrightarrow{t} \sigma'$, *if* $\langle \sigma, t, \sigma' \rangle \in R(\mathbf{p})$

We restrict ourselves to deadlock-free counter systems, i.e., counter systems where the transition relation is total (every configuration has a successor). One way to ensure deadlock-freedom is to require that in every configuration σ, and for every location $\ell \in \mathcal{L}$, it holds that $\sigma \models \bigvee_{r \in \mathcal{R} \wedge r.from = \ell} r.\varphi$. This condition ensures that it is always possible to move out of every location, since there is at least one outgoing rule per location whose guard is satisfied.

Paths and schedules in a counter system. We now define paths and schedules of a counter system, as sequences of configurations and transitions, respectively.

Definition 2. *A* path *in the counter system* $\mathsf{CS}(STA, \mathbf{p}) = (\Sigma(\mathbf{p}), I(\mathbf{p}), R(\mathbf{p}))$ *is a finite sequence* $\{\sigma_i\}_{i=0}^k$ *of configurations, such that for every two consecutive configurations* σ_{i-1}, σ_i, *for* $0 < i \leq k$, *there exists a transition* $t_i \in T(\mathbf{p})$ *such that* $\sigma_{i-1} \xrightarrow{t_i} \sigma_i$. *A path* $\{\sigma_i\}_{i=0}^k$ *is called an* execution *if* $\sigma_0 \in I(\mathbf{p})$.

Definition 3. *A* schedule *is a finite sequence* $\tau = \{t_i\}_{i=1}^k$ *of transitions* $t_i \in T(\mathbf{p})$, *for* $0 < i \leq k$. *We denote by* $|\tau| = k$ *the length of the schedule* τ.

A schedule $\tau = \{t_i\}_{i=1}^k$ *is* feasible *if there is a path* $\{\sigma_i\}_{i=0}^k$ *such that* $\sigma_{i-1} \xrightarrow{t_i} \sigma_i$, *for* $0 < i \leq k$. *We call* σ_0 *the* origin, *and* σ_k *the* goal *of* τ, *and write* $\sigma_0 \xrightarrow{\tau} \sigma_k$.

4 Parameterized Reachability and its Undecidability

We show that the following problem is undecidable in general, by reduction from the halting problem of a two-counter machine (2CM) [25].

Definition 4 (Parameterized Reachability). *Given a formula* φ, *that is, a Boolean combination of counter atoms, and* $STA = (\mathcal{L}, \mathcal{I}, \Pi, \mathcal{R}, RC, \chi)$, *the* parameterized reachability *problem is to decide whether there exists an admissible instance* $\mathbf{p} \in P_{RC}$, *such that in the counter system* $\mathsf{CS}(STA, \mathbf{p})$, *there is an initial configuration* $\sigma \in I(\mathbf{p})$, *and a feasible schedule* τ, *with* $\sigma \xrightarrow{\tau} \sigma'$ *and* $\sigma' \models \varphi$.

To prove undecidability, we construct a synchronous threshold automaton $STA_{\mathcal{M}}$, such that every counter system induced by it simulates the steps of a 2CM executing a program P. The STA has a single parameter – the number n of processes. The idea is that each process plays one of two roles: either it is used to encode the control flow of the program P (*controller* role), or it is used to encode the values of the registers in unary, as in [15] (*storage* role). Thus, $STA_{\mathcal{M}}$ consists of two parts – one for the controller, and one for the storage role.

Our construction allows multiple processes to act as controllers. Since we assume that 2CM is deterministic, all the controllers behave the same. For each instruction of the program P, in the controller part of $STA_{\mathcal{M}}$, there is a single

location (for 'jump if zero' and 'halt') or a pair of locations (for 'increment' and 'decrement'), and a special *stuck* location. In the storage part of $STA_{\mathcal{M}}$, there is a location for each register, a store location, and auxiliary locations. The number of processes in a register location encodes the value of the register in 2CM.

An increment (resp. decrement) of a register is modeled by moving one process from (resp. to) the store location to (resp. from) the register location. The guards on the rules in the controller part check whether the storage processes made a transition that truly models a step of 2CM; if this is the case, the controllers move on to the next location, otherwise they move to the stuck location. For example, to model a 'jump if zero' for register A, the controllers check if $\#\{\ell_A\} = 0$, where ℓ_A is the storage location corresponding to the register A. The main invariant which ensures correctness is that every transition in every counter system induced by $STA_{\mathcal{M}}$, either faithfully simulates a step of the 2CM, or moves all of the controllers to the stuck location.

Let ℓ_{halt} be the halting location in the controller part of $STA_{\mathcal{M}}$. The formula $\varphi \equiv \neg(\#\{\ell_{halt}\} = 0)$ states that the controllers have reached the halting location. Thus, the answer to the parameterized reachability question given the formula φ and $STA_{\mathcal{M}}$ is positive iff 2CM halts, which gives us undecidability.

5 Bounded Diameter Oracle

5.1 Computing the Diameter Using SMT

Intuitively, given an STA, the diameter is the maximal number of transitions needed to reach all possible configurations in every counter system induced by the STA, and an admissible instance $\mathbf{p} \in P_{RC}$. In the following, we adapt the definition of diameter from [10].

Definition 5 (Diameter). *Given an* $STA = (\mathcal{L}, \mathcal{I}, \Pi, \mathcal{R}, RC, \chi)$*, the* diameter *is the smallest number* d *such that for every* $\mathbf{p} \in P_{RC}$ *and every path* $\{\sigma_i\}_{i=0}^{d+1}$ *of length* $d+1$ *in* $\mathsf{CS}(STA, \mathbf{p})$*, there exists a path* $\{\sigma'_j\}_{j=0}^{e}$ *of length* $e \le d$ *in* $\mathsf{CS}(STA, \mathbf{p})$*, such that* $\sigma_0 = \sigma'_0$ *and* $\sigma_{d+1} = \sigma'_e$*.*

Thus, the diameter is the smallest number d that satisfies the formula:

$$\forall \mathbf{p} \in P_{RC}.\, \forall \sigma_0, \dots, \sigma_{d+1}.\, \forall t_1, \dots, t_{d+1}.\, \exists \sigma'_0, \dots, \sigma'_d.\, \exists t'_1, \dots, t'_d.$$
$$Path(\sigma_0, \sigma_{d+1}, d+1) \rightarrow (\sigma_0 = \sigma'_0) \wedge Path(\sigma'_0, \sigma'_d, d) \wedge \bigvee\nolimits_{i=0}^{d} \sigma'_i = \sigma_{d+1} \quad (1)$$

where $Path(\sigma_0, \sigma_d, d)$ is a shorthand for the formula $\bigwedge_{i=0}^{d-1} R(\sigma_i, t_{i+1}, \sigma_{i+1})$, and $R(\sigma, t, \sigma')$ is a predicate which evaluates to true whenever $\sigma \xrightarrow{t} \sigma'$. Since we assume deadlock-freedom, we are able to encode the path $Path(\sigma'_0, \sigma'_d, d)$ of length d, even if the disjunction $\bigvee_{i=0}^{d} \sigma'_i = \sigma_{d+1}$ holds for some $i \le d$.

Formula (1) gives us the following procedure to determine the diameter:

1. initialize the candidate diameter d to 1;

2. check if the negation of the formula (1) is unsatisfiable;
3. if yes, then output d and terminate;
4. if not, then increment d and jump to step 2.

If the procedure terminates, it outputs the diameter, which can be used as completeness threshold for bounded model checking. We implemented this procedure, and used a back-end SMT solver to automate the test in step 2. Our tool generates an SMT-LIB file that encodes the negation of formula (1) as follows.

First, we declare integer constants `n`, `t`, and `f`, corresponding to the parameters, and add an assertion that encodes the resilience condition, e.g., $\mathtt{n} > \mathtt{t} \wedge \mathtt{t} \geq \mathtt{f}$. This ensures that the values assigned to the parameters are admissible.

Second, we encode the formula $Path(\sigma_0, \sigma_{d+1}, d+1)$. We declare integer constants `c_i_j` that correspond to the value of the counter $\kappa[\ell_j]$ in configuration σ_i, for $0 \leq i \leq d+1$, and $0 \leq j < |\mathcal{L}|$, and integer constants `t_k_m` corresponding to the factor of rule r_m in transition t_k, for $0 < k \leq d+1$, and $0 \leq m < |\mathcal{R}|$. We add assertions that every `c_i_j` and `t_k_m` is greater or equal to 0, and additional assertions to ensure that the constants `c_i_j` satisfy the counter invariant χ. Then, we add assertions that model the predicate R:

- for every rule $r_m \in \mathcal{R}$, we assert that its factor $t_k(r_m)$ is 0, if its guard is not satisfied in the configuration σ_{k-1}, for $0 < k \leq d+1$;
- for $0 \leq i < d+1$, we assert that σ_i is the origin and σ_{i+1} is the goal of t_{i+1}.

Finally, we encode the diameter query, which is a universally quantified formula over integer variables `x_i_j`, modeling the value of the counters $\kappa[\ell_j]$ in σ'_i, for $0 \leq i \leq d$ and $0 \leq j < |\mathcal{L}|$, and `y_k_m`, modeling the factors $t'_k(r_m)$, for $0 < k \leq d$ and $0 \leq m < |\mathcal{R}|$. The body of the diameter query is an implication $P \rightarrow Q$, where P is the conjunction of $\bigwedge_{j=0}^{|\mathcal{L}|-1} \mathtt{x_0_j} = \mathtt{c_0_j}$ and of the constraints that encode $Path(\sigma'_0, \sigma'_d, d)$. In Q, we assert that $\sigma'_i \neq \sigma_{d+1}$, for $0 \leq i \leq d$, i.e., $\bigwedge_{i=0}^{d} \bigvee_{j=0}^{|\mathcal{L}|-1} last_j \neq \mathtt{x_i_j}$, where each $last_j$ encodes $\kappa[\ell_j]$ in σ_{d+1}.

5.2 Bounded Diameter for a Fragment of STA

In this section, we show that for a specific fragment of STA, we are able to give a theoretical bound on the diameter.

The STA that fall in this fragment are *monotonic* and *1-cyclic*. An STA is monotonic iff every counter atom changes its truth value at most once in every path of a counter system induced by the STA and an admissible instance $\mathbf{p} \in P_{RC}$. This implies that every schedule can be partitioned into finitely many sub-schedules, that satisfy a property we call *steadiness*. We call a schedule *steady* if the set of rules whose guards are satisfied does not change in all of its transitions. We also give a sufficient condition for monotonicity, using *trapped* counter atoms, defined below. In a 1-cyclic STA, the only cycles that can be formed by its rules are self-loops. Under these two conditions, we guarantee that for every steady schedule, there exists a steady schedule of bounded length, that has the same origin and goal. In the remainder of this section, we show that this

bound depends on the counter atoms Ψ occurring in the guards of the STA, and the length longest path in the STA, denoted by c. The main result of this section is stated by the following theorem:

Theorem 1. *For every feasible schedule τ in a counter system $\mathsf{CS}(STA, \mathbf{p})$, where STA is monotonic and 1-cyclic, and $\mathbf{p} \in P_{RC}$, there exists a feasible schedule τ' of length $O(|\Psi|c)$, such that τ and τ' have the same origin and goal.*

To prove Theorem 1, we start by defining monotonic STA.

Definition 6 (Monotonic STA). *An automaton $STA = (\mathcal{L}, \mathcal{I}, \Pi, \mathcal{R}, RC, \chi)$ is monotonic iff for every path $\{\sigma_i\}_{i=0}^{k}$ in the counter system $\mathsf{CS}(STA, \mathbf{p})$, for $\mathbf{p} \in P_{RC}$, and every counter atom $\psi \in \Psi$, we have $\sigma_i \models \psi$ implies $\sigma_j \models \psi$, for $0 \leq i < j < k$.*

To show that we can partition a schedule into finitely many sub-schedules, we need the notion of a context. A *context* of a transition $t \in T(\mathbf{p})$ is the set $\mathcal{C}_t = \{\psi \in \Psi \mid o(t) \models \psi\}$ of counter atoms ψ satisfied in the origin $o(t)$ of the transition t. Given a feasible schedule τ, the point i is a *context switch*, if $\mathcal{C}_{t_{i-1}} \neq \mathcal{C}_{t_i}$, for $1 < i \leq |\tau|$.

Lemma 1. *Every feasible schedule τ in a counter system induced by a monotonic STA has at most $|\Psi|$ context switches.*

Proof. Let $\tau = \{t_i\}_{i=1}^{k}$ be a feasible schedule and Ψ the set of counter atoms appearing on the rules of the monotonic STA. For every $\psi \in \Psi$, there is at most one context switch i, for $0 < i \leq k$, such that $\psi \notin \mathcal{C}_{t_{i-1}}$ and $\psi \in \mathcal{C}_{t_i}$. □

Sufficient condition for monotonicity. We now introduce the notion of traps.

Definition 7. *A set $L \subseteq \mathcal{L}$ of locations is called a* trap*, iff for every $\ell \in L$ and every $r \in \mathcal{R}$ such that $\ell = r.from$, it holds that $r.to \in L$.*

A counter atom $\psi \equiv \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$ is trapped *iff the set L is a trap.*

The following lemma states that, once a trapped counter atom holds in a configuration, it also holds in its immediate successor.

Lemma 2. *Let $\psi \equiv \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$ be a trapped counter atom, σ a configuration such that $\sigma \models \psi$, and t a transition enabled in σ. If $\sigma \xrightarrow{t} \sigma'$, then $\sigma' \models \psi$.*

Corollary 1. *Let $STA = (\mathcal{L}, \mathcal{I}, \Pi, \mathcal{R}, RC, \chi)$ be an automaton such that all its counter atoms are trapped. Then STA is monotonic.*

Steady schedules. We now define the notion of steadiness, similarly to [18].

Definition 8. *A schedule $\tau = \{t_i\}_{i=1}^{k}$ is* steady*, if $\mathcal{C}_{t_i} = \mathcal{C}_{t_j}$, for $0 < i < j \leq k$.*

We now focus on shortening steady schedules. That is, given a steady schedule, we construct a schedule of bounded length with the same origin and goal.

Observe that $STA = (\mathcal{L}, \mathcal{I}, \Pi, \mathcal{R}, RC, \chi)$ can be seen as a directed graph G_{STA}, with vertices corresponding to the locations $\ell \in \mathcal{L}$, and edges corresponding to the rules $r \in \mathcal{R}$. We call the *longest chain* c of STA the length of the longest path between two nodes in the graph G_{STA}. If G_{STA} contains only cycles of length one, then STA is called *1-cyclic*.

To shorten steady schedules, in addition to monotonicity, we require that the STA are also 1-cyclic. In the following, we assume that the schedules we shorten come from counter systems induced by monotonic and 1-cyclic STA. Intuitively, if a given schedule is longer than the longest chain of the STA, then in some transition of the schedule some processes followed a rule which is a self-loop. As processes may follow self-loops at different transitions, we cannot shorten the given schedule by eliminating transitions as a whole. Instead, we deconstruct the original schedule into sequences of process steps, which we call *runs*, shorten the runs, and reconstruct a new shorter schedule from the shortened runs. The main challenge is to show that the newly obtained schedule is feasible and steady.

Schedules as multisets of runs. We proceed by defining runs and showing that each schedule can be represented by a multiset of runs.

We call a *run* the sequence $\varrho = \{r_i\}_{i=1}^{k}$ of rules, for $r_i \in \mathcal{R}$, such that $r_i.to = r_{i+1}.from$, for $0 < i < k$. We denote by $\varrho[i] = r_i$ the i-th rule in the run ϱ, and by $|\varrho|$ the length of the run. The following lemma shows that a feasible schedule can be deconstructed into a multiset of runs.

Lemma 3. *For every feasible schedule $\tau = \{t_i\}_{i=1}^{k}$, there exists a multiset $(\mathcal{P}, m)$, where*

1. *$\mathcal{P}$ is a set of runs ϱ of length k, and*
2. *$m : \mathcal{P} \to \mathbb{N}$ is a multiplicity function, such that for every location $\ell \in \mathcal{L}$, it holds that $\sum_{r.from=\ell} t_i(r) = \sum_{\varrho[i].from=\ell} m(\varrho)$, for $0 < i \leq k$.*

A multiset $(\mathcal{P}, m)$ of runs of length k defines a schedule $\tau = \{t_i\}_{i=1}^{k}$ of length k, and we have $t_i(r) = \sum_{\varrho[i]=r} m(\varrho)$, for every rule $r \in \mathcal{R}$ and $0 < i \leq k$.

For the counter systems of STA, which are both monotonic and 1-cyclic, we show that their steady schedules can be shortened, so that their length does not exceed the longest chain c in the automaton.

Lemma 4. *Let τ be a steady feasible schedule in a counter system induced by a monotonic and 1-cyclic STA. If $|\tau| > c + 1$, then there exists a steady feasible schedule τ' such such that $|\tau'| = |\tau| - 1$, and τ, τ' have the same origin and goal.*

Proof (Sketch). Observe that if $\tau = \{t_i\}_{i=1}^{k+1}$, with $|\tau| = k+1 > c+1$, is a steady schedule, then $\mathcal{C}_{t_1} = \mathcal{C}_{t_k}$, and its prefix $\theta = \{t_i\}_{i=1}^{k}$ is a steady and feasible schedule, with $k > c$. By Lemma 3, there is a multiset $(\mathcal{P}, m)$ of runs of length k describing θ. Since $k > c$, and c is the longest chain in the STA, which is 1-cyclic, it must be the case that every run in $\mathcal{P}$ contains at least one self-loop.

Construct a new multiset $(\mathcal{P}', m')$ of runs of length $k-1$, such that each $\varrho' \in \mathcal{P}'$ is obtained by some $\varrho \in \mathcal{P}$ by removing one occurrence of a self-loop rule. The multiset $(\mathcal{P}', m')$ defines the schedule $\theta' = \{t'_1\}_{i=1}^{k-1}$. Since we only remove self-loops (which go from and to the same location) when we build θ' from θ, the feasibility is preserved. Furthermore, it is easy to check that θ' has the same origin and goal as θ. As the goal of θ' is the origin of t_{k+1}, construct a schedule $\tau' = \{t'_i\}_{i=1}^{k}$, where $t'_k = t_{k+1}$. The steadiness of τ' follows from the steadiness of τ and the fact that $o(t_1) = o(t'_1)$ and $o(t_{k+1}) = o(t'_k)$. □

As a consequence of Lemma 1 and 4, we obtain Theorem 1, which tells us that for any feasible schedule, there exists a feasible schedule of length $O(|\Psi|c)$. This bound does not depend on the parameters, but on the number of context switches and the longest chain c, which are properties of the STA.

6 Bounded Model Checking of Safety Properties

Once we obtain the diameter bound d (either using the procedure from Section 5.1, or by Theorem 1), we use it as a completeness threshold for bounded model checking. For the algorithms that we verify, we express the violations of their safety properties as reachability queries on bounded executions. The length of the bounded executions depends on d, and on the environment assumption of the algorithm, which determines if there is a clean round in every execution.

Checking Safety for Algorithms without a Clean Round Assumption. Here, we search for violations of safety properties in executions of length $e \leq d$, by checking satisfiability of the formula:

$$\exists \mathbf{p} \in P_{RC}.\, \exists \sigma_0, \dots, \sigma_e.\, \exists t_1, \dots, t_e.\ \mathit{Init}(\sigma_0) \wedge \mathit{Path}(\sigma_0, \sigma_e, e) \wedge \mathit{Bad}(\sigma_e) \quad (2)$$

where the predicate $\mathit{Init}(\sigma)$ encodes that σ is an initial configuration, together with the constraints imposed on the initial configuration by the safety property, and $\mathit{Bad}(\sigma)$ encodes the bad configuration, which, if reachable, violates safety.

For example, the algorithm from Figure 1 has to satisfy the safety property *unforgeability*: If no process sets `v` to 1 initially, then no process ever sets `accept` to true. In our encoding, we check executions of length $e \leq d$, whose initial configuration has the counter $\kappa[\text{V1}] = 0$. In a bad configuration, the counter $\kappa[\text{AC}] > 0$. Thus, to find violations of unforgeability, in formula (2), we set:

$$\mathit{Init}(\sigma_0) \equiv \sigma_0.\kappa[\text{V0}] + \sigma_0.\kappa[\text{V1}] = N(\mathbf{p}) \wedge \sigma_0.\kappa[\text{V1}] = 0$$
$$\mathit{Bad}(\sigma_e) \equiv \sigma_e.\kappa[\text{AC}] > 0$$

Checking Safety for Algorithms with a Clean Round. We check for violations of safety in executions of length $e \leq 2d+1$, where $e = e_1 + e_2 + 1$ such that: (i) we find all reachable clean-round configurations in an execution of length e_1+1, for $e_1 \leq d$, such that the last configuration σ_{e_1+1} satisfies the clean round

condition, and (ii) we check if a bad configuration is reachable from σ_{e_1+1} by a path of length $e_2 \leq d$. That is, we check satisfiability of the formula:

$$\exists \mathbf{p} \in P_{RC}.\, \exists \sigma_0, \ldots, \sigma_e.\, \exists t_1, \ldots, t_e.\ \mathit{Init}(\sigma_0) \wedge \mathit{Path}(\sigma_0, \sigma_{e_1+1}, e_1 + 1) \\ \wedge\, \mathit{Clean}(\sigma_{e+1}) \wedge \mathit{Path}(\sigma_{e_1+1}, \sigma_e, e_2) \wedge \mathit{Bad}(\sigma_e) \quad (3)$$

where the predicate $\mathit{Clean}(\sigma)$ encodes the clean round condition.

For example, one of the safety properties that the *FloodMin* algorithm for $k = 1$, from Figure 2 has to satisfy, is *k-agreement*, which requires that at most k different values are decided. In the original algorithm, the processes decide after $\lfloor t/k \rfloor + 1$ rounds, such that at least one of them is the clean round, in which $k - 1$ processes crash. In our encoding, we check paths of length $e \leq 2d + 1$. We enforce the clean round by asserting that the counters of the locations $\mathrm{C0}, \mathrm{C1}$ are $k - 1 = 0$ in the configuration σ_{e_1+1}. Agreement is violated if in the last configuration σ_e the counters $\kappa[\mathrm{V0}]$ and $\kappa[\mathrm{V1}]$ are non-zero. That is, to check agreement, in formula (3) we set:

$$\begin{aligned} \mathit{Init}(\sigma_0) &\equiv \sigma_0.\kappa[\mathrm{V0}] + \sigma_0.\kappa[\mathrm{V1}] + \sigma_0.\kappa[\mathrm{C0}] + \sigma_0.\kappa[\mathrm{C1}] = N(\mathbf{p}) \\ \mathit{Clean}(\sigma_{e_1+1}) &\equiv \sigma_{e_1+1}.\kappa[\mathrm{C0}] + \sigma_{e_1+1}.\kappa[\mathrm{C1}] = 0 \\ \mathit{Bad}(\sigma_e) &\equiv \sigma_e.\kappa[\mathrm{V0}] > 0 \wedge \sigma_e.\kappa[\mathrm{V1}] > 0 \end{aligned}$$

7 Experimental evaluation

The algorithms that we model using STA and verify by bounded model checking are designed for different fault models, which in our case are crashes, send omissions or Byzantine faults. We now proceed by introducing our benchmarks.

Algorithms without a Clean Round Assumption. We consider three variants of the synchronous reliable broadcast algorithm, whose STA are monotonic and 1-cyclic. The three algorithms assume different fault models:

- `rb`, [28] (Figure 1): reliable broadcast with at most t Byzantine faults;
- `rb_hybrid`, [9]: reliable broadcast with hybrid faults: at most t_b Byzantine and at most t_o send omissions;
- `rb_omit`, [9]: reliable broadcast with at most t.

Algorithms with a Clean Round. We encode several algorithms from this class, that solve the consensus or k-set agreement problem:

- `fair_cons` [27], `floodset` [23]: consensus with crash faults;
- `floodmin`, for $k \in \{1, 2\}$ [23] (Figure 2): k-set agreement with crash faults;
- `kset_omit`, for $k \in \{1, 2\}$ [27]: k-set agreement with send omission faults;
- `phase_king` [7, 8], `phase_queen` [7]: consensus with Byzantine faults.

These algorithms have a structure similar to the one described in Section 2, with the exception of `phase_king` and `phase_queen`. Their loop body consists of several message exchange steps, which correspond to multiple rounds, grouped in a *phase*. In each phase, a designated process acts as a coordinator.

Table 2. Results for our benchmarks, available at [1]: $|\mathcal{L}|, |\mathcal{R}|, |\Psi|$, RC are the number of locations, rules, atomic guards, and resilience condition in each STA; d is the diameter computed using SMT, c is the longest chain of the algorithms whose STA are monotonic and 1-cyclic; t_d is the time (in seconds) to compute the diameter using SMT; t_b, for `len` $\sim d$, is the time to check reachability using the diameter computed using the SMT procedure from Section 5.1; t_b, for `len` $\sim |\Psi|c$ the time to check reachability using the bound obtained by Theorem 1. The experiments were run on a machine with Intel(R) Core(TM) i5-4210U CPU and 4GB of RAM, using `z3-4.8.1` and `cvc4-1.6`.

algorithm	$\lvert\mathcal{L}\rvert$	$\lvert\mathcal{R}\rvert$	$\lvert\Psi\rvert$	RC	d	t_d		t_b, `len` $\sim d$		c	t_b, `len` $\sim \lvert\Psi\rvert c$	
						z3	cvc4	z3	cvc4		z3	cvc4
`rb`	4	8	4	$n > 3t$	2	0.41	1.01	0.16	0.14	2	0.50	1.25
`rb_hybrid`	8	16	4	$n > 2t$	2	1.31	37.98	0.18	0.19	2	1.14	1.93
`rb_omit`	8	16	4	$n > 3t_b + 2t_o$	2	0.51	2.57	0.17	0.19	2	0.79	1.82
`fair_cons`	11	20	2	$n > t$	2	0.88	10.27	0.36	0.87	–	–	–
`floodmin`, $k = 1$	5	8	2	$n > t$	2	0.32	1.09	0.33	0.64	–	–	–
`floodmin`, $k = 2$	7	16	4	$n > t$	2	0.65	7.37	0.38	0.81	–	–	–
`floodset`	7	14	4	$n > t$	2	0.44	3.18	0.35	0.73	–	–	–
`kset_omit`, $k = 1$	4	6	2	$n > t$	1	0.18	0.17	0.08	0.09	–	–	–
`kset_omit`, $k = 2$	6	12	4	$n > t$	1	0.27	0.33	0.11	0.19	–	–	–
`phase_king`	34	72	12	$n > 3t$	4	15.28	50.53	4.35	9.77	–	–	–
`phase_queen`	24	42	8	$n > 4t$	3	1.66	17.74	0.90	3.50	–	–	–

Computing the diameter. We implemented the procedure from Section 5.1 in Python. The implementation uses a back-end SMT solver (currently, `z3` and `cvc4`). Our tool computed diameter bounds for all of our benchmarks, even for those for which we do not have a theoretical guarantee. Our experiments reveal extremely low values for the diameter, that range between 1 and 4. The values for the diameter and the time needed to compute them are presented in Table 2.

Checking the algorithms. We have implemented another Python function which encodes violations of the safety properties as reachability properties on paths of bounded length, as described in Section 6, and uses a back-end SMT solver to check their satisfiability. Table 2 contains the results that we obtained by checking reachability for our benchmarks, using the diameter bound computed using the procedure from Section 5.1, and diameter bound from Theorem 1, for algorithms whose STA are monotonic and 1-cyclic.

When checking the bounded executions of `phase_king` and `phase_queen`, we have to enforce a *clean phase*, rather than clean round. We easily adapt the method presented in Section 6 to check executions of length $\leq 2d + p$ for these algorithms, where p is the number of rounds per phase.

To our knowledge, we are the first to verify the listed algorithms that work with send omission, Byzantine and hybrid faults. For the algorithms with crash faults, our approach is a significant improvement to the results obtained using the abstraction-based method from [3].

Counterexamples. Our tool found a bug in the version of the `phase_king` algorithm that was given in [7], which was corrected in the version of the algorithm in [8]. The version from [7] had the wrong threshold '$> n - t$' in one guard,

while the one in [8] had '$\geq n - t$' for the same guard. To test our tool, we produced erroneous encodings for our benchmarks, and checked them. For `rb`, `rb_hybrid`, `rb_omit`, `phase_king`, and `phase_queen`, we tweaked the resilience condition, and introduced more faults than expected by the algorithm, e.g., by setting $f > t$ (instead of $f \leq t$) in the STA in Figure 1. For `fair_cons`, `floodmin`, `floodset`, and `kset_omit`, we checked executions without a clean round. For all of the erroneous encodings, our tool produces counterexamples in seconds.

8 Discussion and Related Work

Parameterized verification of synchronous and partially synchronous distributed algorithms has recently gained attention. Both models have in common that distributed computations are organized in rounds and processes (conceptually) move in lock-step. For partially synchronous consensus algorithms, the authors of [13] introduced a consensus logic and (semi-)decision procedures. Later, the authors of [24] introduced a language for partially synchronous consensus algorithms, and proved cut-off theorems specialized to the properties of consensus: agreement, validity, and termination. Concerning synchronous algorithms, the authors of [3] introduced an abstraction-based model checking technique for crash-tolerant synchronous algorithms with existential guards. In contrast to their work, we allow more general guards that contain linear expressions over the parameters, e.g., $n - t$. Our method offers more automation, and our experimental evaluation shows that our technique is faster than the technique [3].

Our formal framework is a *synchronous* variant of threshold automata, which were introduced in [19] for asynchronous fault-tolerant distributed algorithms. Several extensions of this model were recently studied in [21], but the synchronous case was not considered. From a theoretical viewpoint, synchronous threshold automata extend the guarded protocols by Emerson and Namjoshi [14], in which a process can check only if a sum of counters is different from 0 or n. Our synchronous threshold automata generalize their existential and universal guards by allowing more fine-grained counting. It is not straightforward whether the results from [14] can be generalized to synchronous threshold automata.

In [2], safety of finite-state transition systems over infinite data domains, that satisfy certain additional properties was reduced to backwards reachability checking, using a fixpoint computation. It would be interesting to put our results in this context. FAST [5] accelerates the transition relation of a counter systems to compute a set of reachable configurations. We found that FAST can handle the algorithms without a clean round. However, it fails to compute a fixpoint for the algorithms that require a clean round and whose executions in general depend on the parameters.

Our experiments show that STA that are neither monotonic, nor 1 cyclic still may have bounded diameters. Finding other classes of STA for which one could derive the diameter bounds is a subject of future work. Although we considered only reachability properties in this work — which happened to be challenging — we are going to investigate completeness thresholds for liveness in the future.

References

1. Experiments. https://github.com/istoilkovska/syncTA
2. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.: General decidability theorems for infinite-state systems. In: LICS. pp. 313–321 (1996)
3. Aminof, B., Rubin, S., Stoilkovska, I., Widder, J., Zuleger, F.: Parameterized model checking of synchronous distributed algorithms by abstraction. In: VMCAI. LNCS, vol. 10747, pp. 1–24. Springer (2018)
4. Attiya, H., Welch, J.: Distributed Computing. Wiley, 2nd edn. (2004)
5. Bardin, S., Leroux, J., Point, G.: Fast extended release. In: CAV. pp. 63–66 (2006)
6. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV. pp. 171–177 (2011)
7. 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
8. Berman, P., Garay, J.A., Perry, K.J.: Towards optimal distributed consensus (extended abstract). In: FOCS. pp. 410–415 (1989)
9. Biely, M., Schmid, U., Weiss, B.: Synchronous consensus under hybrid process and link failures. Theor. Comput. Sci. 412(40), 5602–5630 (2011)
10. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: TACAS. LNCS, vol. 1579, pp. 193–207 (1999)
11. Chaudhuri, S., Herlihy, M., Lynch, N.A., Tuttle, M.R.: Tight bounds for k-set agreement. J. ACM 47(5), 912–943 (2000)
12. Clarke, E.M., Kroening, D., Ouaknine, J., Strichman, O.: Completeness and complexity of bounded model checking. In: VMCAI. LNCS, vol. 2937, pp. 85–96 (2004)
13. 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)
14. Emerson, E.A., Namjoshi, K.S.: Automatic verification of parameterized synchronous systems. In: CAV, LNCS, vol. 1102, pp. 87–98. Springer (1996)
15. Emerson, E.A., Namjoshi, K.S.: On reasoning about rings. Int. J. Found. Comput. Sci. 14(4), 527–550 (2003)
16. 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)
17. 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)
18. 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)
19. 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), http://dx.doi.org/10.1016/j.ic.2016.03.006
20. Kroening, D., Strichman, O.: Efficient computation of recurrence diameters. In: VMCAI. LNCS, vol. 2575, pp. 298–309 (2003)
21. Kukovec, J., Konnov, I., Widder, J.: Reachability in parameterized systems: All flavors of threshold automata. In: CONCUR. pp. 19:1–19:17 (2018)
22. 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)
23. Lynch, N.: Distributed Algorithms. Morgan Kaufman (1996)

24. Marić, O., Sprenger, C., Basin, D.A.: Cutoff Bounds for Consensus Algorithms. In: CAV. pp. 217–237 (2017)
25. Minsky, M.L.: Computation: Finite and Infinite Machines. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1967)
26. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS. pp. 337–340 (2008)
27. Raynal, M.: Fault-tolerant Agreement in Synchronous Message-passing Systems. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers (2010)
28. Srikanth, T.K., Toueg, S.: Optimal clock synchronization. Journal of the ACM 34(3), 626–645 (1987)
29. Srikanth, T., Toueg, S.: Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. Dist. Comp. 2, 80–94 (1987)

Appendix

A Undecidability of parameterized reachability

We show that the parameterized reachability problem is undecidable in general, by reduction from the halting problem of a two-counter machine [25].

A two-counter machine (2CM) $\mathcal{M}$ consists of two *registers* A and B, and a set $\mathcal{I} = \{inc_i, dec_i, zero_i(k), halt \mid i \in \{A, B\} \text{ and } k \in \mathbb{N}\}$ of *instructions*, consisting of increment, decrement and zero test instructions for each register, together with a halting instruction for the machine. A sequence $P = \langle inst_1, \dots, inst_m \rangle$, for $inst_1, \dots, inst_m \in \mathcal{I}$, and $m \in \mathbb{N}$, is called the *program* of $\mathcal{M}$. The machine $\mathcal{M}$ starts the execution of the program P in the initial instruction $inst_1$, and proceeds as follows. If the instruction $inst_j$, for $1 \leq j \leq m$ is an inc_i (resp. dec_i) instruction, it increments (resp. decrements) the register i, for $i \in \{A, B\}$, and moves the control of the program to location $j+1$. If $inst_j$ is a $zero_i(k)$ instruction, it moves the control of the program to the location k in case the register i contains the value 0, for $i \in \{A, B\}$, and to the location $j+1$ otherwise. We require that $inst_m = halt$ is the *halting* instruction of P. We assume that initially the registers A, B contain the value 0.

The halting problem of 2CM is known to be undecidable [25]. We show the undecidability of parameterized reachability by simulating the steps of a 2CM $\mathcal{M}$ takes when executing the program P, using a counter system $\mathsf{CS}(STA_{\mathcal{M}}, \mathbf{p}_{\mathcal{M}})$, where $STA_{\mathcal{M}} = (\mathcal{L}_{\mathcal{M}}, \mathcal{I}_{\mathcal{M}}, \Pi_{\mathcal{M}}, \mathcal{R}_{\mathcal{M}}, RC_{\mathcal{M}}, \chi_{\mathcal{M}})$ encodes the instructions from the program P, and $\mathbf{p}_{\mathcal{M}}$ is an admissible instance of the parameter vector $\boldsymbol{\pi}_{\mathcal{M}} = \langle n \rangle$, containing a single parameter – the number n of processes. The idea is that each of the n processes plays one of two roles in the counter system. That is, each process is either used to encode the control flow of the program P, or it is used to encode the values of the registers in unary [15]. The former are called the *controller* processes and the latter the *storage* processes.

We now proceed by defining the structure of the automaton $STA_{\mathcal{M}}$. The set $\mathcal{L}_{\mathcal{M}} = \mathcal{L}_C \cup \mathcal{L}_S$ of locations is partitioned into locations $\mathcal{L}_C$ and $\mathcal{L}_S$ of the controller and of the storage, respectively. The set $\mathcal{L}_C$ of locations of the controller consists of a location ℓ_j for each instruction $inst_j$, where $1 \leq j \leq m$, and an additional location ℓ_j^* if $inst_j$ is an increment or decrement instruction, and a special location ℓ_{stuck} that denotes a stuck configuration of the 2CM. The set $\mathcal{L}_S$ of locations of the storage contains the location ℓ_{store}, one location for each of the registers ℓ_A, ℓ_B, one location per increment/decrement instruction $\ell_{inc_i}, \ell_{dec_i}$, for $i \in \{A, B\}$. Intuitively, the state ℓ_{store} is used to store processes that will eventually make transitions to one of ℓ_i, for $i \in \{A, B\}$, via ℓ_{inc_i}, and those that make transitions from ℓ_i via ℓ_{dec_i}.

The set $\mathcal{I}_{\mathcal{M}} \subseteq \mathcal{L}_{\mathcal{M}}$ of initial locations contains the states $\ell_1 \in \mathcal{L}_C$ and $\ell_{store} \in \mathcal{L}_S$.

The set $\Pi_{\mathcal{M}}$ of parameters contains the single parameter n, denoting the number of processes. As there are no other parameters, the resilience condition $RC_{\mathcal{M}} = true$, hence every $\mathbf{p}_{\mathcal{M}}[n] \in \mathbb{N}$ is admissible.

The set $\mathcal{R}_{\mathcal{M}} = \mathcal{R}_C \cup \mathcal{R}_S$ of rules consists of rules $\mathcal{R}_C$ and $\mathcal{R}_S$ for the controller and storage processes, respectively. An increment (resp. decrement) of register A is modeled by moving a single storage process to (resp. from) the location ℓ_A from (resp. to) the location ℓ_{store}, and moving the controller processes to the location corresponding to the next instruction in two steps. That is, the controller processes move from ℓ_j to ℓ_{j+1} via the additional location ℓ_j^*, if $inst_j$ is an increment (resp. decrement) instruction. The zero test of register A is modeled by moving the controller process to the location ℓ_k if there are no processes in the location ℓ_A, and to the location corresponding to the next instruction otherwise. The increment, decrement, and zero test of register B are modeled in an analogous way.

We now formally define the rules in $\mathcal{R}_{\mathcal{M}} = \mathcal{R}_C \cup \mathcal{R}_S$. Let $inst_j$, for $1 \leq j \leq m$, be an instruction of the program P, and $i \in \{A, B\}$. For convenience, we use the notation "$\ell \to \ell'$ if φ" for the rule (ℓ, ℓ', φ).

The set $\mathcal{R}_C$ of rules of the controller contains:

$$\ell_{stuck} \to \ell_{stuck} \qquad \text{if } true \tag{4}$$

$$\ell_m \to \ell_m \qquad \text{if } true \tag{5}$$

Depending on $inst_j$, we consider the following cases.

Case 1. If $inst_j$ is an inc_i instruction, then $\mathcal{R}_C$ contains the rules:

$$\ell_j \to \ell_j^* \qquad \text{if } true \tag{6}$$

$$\ell_j^* \to \ell_{j+1} \qquad \text{if } \#\{\ell_{inc_i}\} = 1 \tag{7}$$

$$\ell_j^* \to \ell_{stuck} \qquad \text{if } \#\{\ell_{inc_i}\} \neq 1 \tag{8}$$

Case 2. If $inst_j$ is a dec_i instruction, then $\mathcal{R}_C$ contains the rules:

$$\ell_j \to \ell_j^* \qquad \text{if } true \tag{9}$$

$$\ell_j^* \to \ell_{j+1} \qquad \text{if } \#\{\ell_{dec_i}\} = 1 \tag{10}$$

$$\ell_j^* \to \ell_{stuck} \qquad \text{if } \#\{\ell_{dec_i}\} \neq 1 \tag{11}$$

Case 3. If $inst_j$ is a $zero_i(k)$ instruction, for $1 \leq k \leq m$, then $\mathcal{R}_{\mathcal{M}}$ contains the rules:

$$\ell_j \to \ell_k \qquad \text{if } \#\{\ell_i\} = 0 \tag{12}$$

$$\ell_j \to \ell_{j+1} \qquad \text{if } \#\{\ell_i\} \neq 0 \tag{13}$$

The set $\mathcal{R}_S$ of rules of the storage contains:

$$\ell_{store} \to \ell_{store} \qquad \text{if } true \tag{14}$$

$$\ell_i \to \ell_i \qquad \text{if } true \tag{15}$$

Again, based on $inst_j$, we consider the cases:

Case 1. If $inst_j$ is an inc_i instruction, then $\mathcal{R}_S$ contains the rules:

$$\ell_{store} \to \ell_{inc_i} \qquad \text{if } \#\{\ell_j\} > 0 \tag{16}$$
$$\ell_{inc_i} \to \ell_i \qquad \text{if } true \tag{17}$$

Case 2. If $inst_j$ is a dec_i instruction, then $\mathcal{R}_S$ contains the rules:

$$\ell_i \to \ell_{dec_i} \qquad \text{if } \#\{\ell_j\} > 0 \tag{18}$$
$$\ell_{dec_i} \to \ell_{store} \qquad \text{if } true \tag{19}$$

Note that, in case $inst_j$ is a $zero_i(k)$ instruction, we do not need to introduce new rules in $\mathcal{R}_S$, as the 2CM does not modify the value of the registers when performing a $zero_i(k)$ instruction.

This construction allows multiple processes to act as controllers, and since we assume that the 2CM is deterministic, all the controllers behave the same. The main invariant which ensures correctness is that every transition in a counter system induced by $STA_{\mathcal{M}}$ either faithfully simulates a step of the 2CM, or moves all of the controller processes to the stuck location. Furthermore, if there are no controller processes in the stuck location, the number of processes in locations ℓ_A, ℓ_B denote the current values of the registers A, B, respectively.

To truly model an increment (resp. decrement) of register i, for $i \in \{A, B\}$, the controller processes have to ensure that exactly one process was moved to (resp. from) the location ℓ_i to the location ℓ_{inc_i} (resp. ℓ_{dec_i}). In other words, for every inc_i (resp. dec_i) instruction in the program P, for $i \in \{A, B\}$, if the guard of rule (7) (resp. (10)) is satisfied, the controllers move to the location corresponding to the next instruction, and the number of storage processes in the location ℓ_i is increased (resp. decreased) by exactly one. Otherwise, all controller processes are moved to the stuck location, and the number of processes in ℓ_i no longer correspond to the value of register i. Similarly, for every $zero_i(k)$ instruction, all the controllers move to the location ℓ_k if the guard of rule (12) is satisfied, that is, if the number of storage processes in the location ℓ_i is 0, which corresponds to the value of register i being equal to 0. Otherwise, the controllers move to the location corresponding to the next instruction in the program P.

The formula $\varphi \equiv \#\{\ell_m\} \neq 0$ states that the controller processes reach the location $\ell_m \in \mathcal{L}_C$, which corresponds to the halting instruction $inst_m = halt$ of the program P.

If $\mathcal{M}$ halts, by the above construction we get that for some $\mathbf{p}_{\mathcal{M}}[n] \in \mathbb{N}$, there is a configuration σ' in the counter system $\mathsf{CS}(STA_{\mathcal{M}}, \mathbf{p}_{\mathcal{M}})$, reachable from the initial configuration σ, such that φ holds in σ', i.e., $\sigma' \models \varphi$.

In the other direction, if $\mathcal{M}$ does not halt, then we have that for every $\mathbf{p}_{\mathcal{M}}[n] \in \mathbb{N}$, and every configuration σ' in the counter system $\mathsf{CS}(STA_{\mathcal{M}}, \mathbf{p}_{\mathcal{M}})$, reachable from the initial configuration σ, we have $\sigma' \not\models \varphi$.

Thus, the answer to the parameterized reachability question given the formula φ and the synchronous threshold automaton $STA_{\mathcal{M}}$ is positive iff $\mathcal{M}$ halts, which gives us undecidability of parameterized reachability.

B Detailed Proofs

The following proposition states a property of feasible schedules, and is a consequence of Definition 3.

Proposition A *The schedule $\tau = \{t_i\}_{i=1}^k$ is feasible iff for every $\ell \in \mathcal{L}$, we have: $\sum_{r.to=\ell} t_{i-1}(r) = \sum_{r'.from=\ell} t_i(r')$ for $r, r' \in \mathcal{R}$ and $1 < i \leq k$.*

The following lemma states a property of trapped counter atoms.

Lemma 2. *Let $\psi \equiv \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$ be a trapped counter atom, σ a configuration such that $\sigma \models \psi$, and t a transition enabled in σ. If $\sigma \xrightarrow{t} \sigma'$, then $\sigma' \models \psi$.*

Proof. Suppose that $\sigma \models \psi$, that is, $\sum_{\ell \in L} \sigma.\kappa[\ell] \geq \boldsymbol{a} \cdot \sigma.\mathbf{p} + b$. Because L is a trap, by Definition 1 and 7, we have

$$\sum_{\ell \in L} \sigma'.\kappa[\ell] = \sum_{r \in \mathcal{R} \wedge r.to \in L} t(r) \geq \sum_{r \in \mathcal{R} \wedge r.from \in L} t(r) = \sum_{\ell \in L} \sigma.\kappa[\ell]$$

Thus, $\sum_{\ell \in L} \sigma'.\kappa[\ell] \geq \boldsymbol{a} \cdot \sigma'.\mathbf{p} + b$, and $\sigma' \models \psi$. □

The following proposition states a property of steady schedules in counter systems induced by monotonic STA.

Proposition B *Let $STA = (\mathcal{L}, \mathcal{I}, \Pi, \mathcal{R}, RC, \chi)$ be monotonic, $\mathbf{p} \in P_{RC}$, and $\tau = \{t_i\}_{i=1}^k$ a schedule in $\mathsf{CS}(STA, \mathbf{p})$. If $\mathcal{C}_{t_1} = \mathcal{C}_{t_k}$, then τ is a steady schedule.*

Proof. Follows from the monotonicity and Definition 8. □

We now present the proof of Lemma 3.

Lemma 3. *For every feasible schedule $\tau = \{t_i\}_{i=1}^k$, there exists a multiset $(\mathcal{P}, m)$, where*

1. *$\mathcal{P}$ is a set of runs ϱ of length k, and*
2. *$m : \mathcal{P} \to \mathbb{N}$ is a multiplicity function, such that for every location $\ell \in \mathcal{L}$, it holds that $\sum_{r.from=\ell} t_i(r) = \sum_{\varrho[i].from=\ell} m(\varrho)$, for $0 < i \leq k$.*

Proof. We proceed by induction on the length of the schedule.

In the induction base, let τ be a schedule of length one, that is, τ consists of a single transition t_1. We define the multiset $(\mathcal{P}, m)$ by setting $\mathcal{P} = \{r \in \mathcal{R} \mid t_1(r) \neq 0\}$ to be the set of rules that have non-zero factors in t_1, and for every $r \in \mathcal{P}$, we set $m(r) = t_1(r)$.

In the induction step, consider a schedule $\tau = \{t_i\}_{i=1}^{k+1}$ of length $k+1$. For the prefix $\tau' = \{t_i\}_{i=1}^k$ of τ, which is a feasible schedule of length k, we have, by the induction hypothesis, that there exists a multiset $(\mathcal{P}', m')$ such that $\mathcal{P}'$ is a set of runs of length k, and for every location $\ell \in \mathcal{L}$ it holds that $\sum_{r.from=\ell} t_i(r) = \sum_{\varrho'[i].from=\ell} m'(\varrho')$, for $0 < i \leq k$. Let $\sigma_k = g(t_k)$ be the goal configuration of t_k. For every $\ell \in \mathcal{L}$, it holds that $\sigma_k.\kappa[\ell] = \sum_{r.to=\ell} t_k(r)$. Observe that $\sigma_k.\kappa[\ell]$

also represents the number of runs that end in a rule that points to the location $\ell \in \mathcal{L}$, that is,

$$\sum_{r.to=\ell} t_k(r) = \sigma_k.\kappa[\ell] = \sum_{\varrho'[k].to=\ell} m'(\varrho') \tag{20}$$

Given the transition t_{k+1}, let $R_{k+1} = \{r \in \mathcal{R} \mid t_{k+1}(r) \neq 0\}$ be the set of rules that have non-zero factors in t_{k+1}. We define the set $\mathcal{P} = \{\varrho' r \mid \varrho' \in \mathcal{P}', r \in R_{k+1}, \varrho'[k].to = r.from\}$ of runs of length $k+1$, where $\varrho' r$ is the run obtained by appending r to ϱ'. We define the function m that maps runs $\varrho = \varrho' r$ from the set $\mathcal{P}$ such that for every $\ell \in \mathcal{L}$, it holds that $\sum_{\varrho'[k].to=\ell} m'(\varrho') = \sum_{r.from=\ell} m(\varrho' r)$.

We now check that the multiset $(\mathcal{P}, m)$ satisfies the two properties. Clearly, the set $\mathcal{P}$ contains runs of length $k+1$. To show the second property, we use the assumption that τ is a feasible schedule. Hence, for every $\ell \in \mathcal{L}$, it holds that

$$\sum_{r.from=\ell} t_{k+1}(r) = \sum_{r.to=\ell} t_k(r) = \sum_{\varrho'[k].to=\ell} m'(\varrho') = \sum_{r.from=\ell} m(\varrho' r)$$

by Proposition A, (20), and the construction respectively. □

The next lemma states that, for every feasible schedule in a counter system induced by a 1-cyclic STA, and whose length is longer than the longest chain c, every run contains a self-loop.

Lemma A *Let τ be a feasible schedule in a counter system induced by a 1-cyclic STA, and $(\mathcal{P}, m)$ its corresponding multiset of runs of length $|\tau|$. If $|\tau| > c$, then every run $\varrho \in \mathcal{P}$ contains a rule $r \in \mathcal{R}$ such that $r.from = r.to$.*

Proof. Suppose $|\tau| > c$ and that there exists a run $\varrho \in \mathcal{P}$, such that for every $0 < i \leq k$, it holds that $\varrho[i].from \neq \varrho[i].to$. Because c is the longest chain in the 1-cyclic STA, and since $|\varrho| > c$, it must be the case that there exist indices i, j, with $0 < i < j \leq k$, such that $\varrho[i].from = \varrho[j].to$. This implies that the STA contains a cycle which is not a self-loop, which is a contradiction to it being 1-cyclic. □

We give the detailed proof of Lemma 4.

Lemma 4. *Let τ be a steady feasible schedule in a counter system induced by a monotonic and 1-cyclic STA. If $|\tau| > c+1$, then there exists a steady feasible schedule τ' such such that $|\tau'| = |\tau| - 1$, and τ, τ' have the same origin and goal.*

Proof. Let $\tau = \{t_i\}_{i=1}^{k+1}$ be a steady feasible schedule, such that $|\tau| = k+1 > c+1$. Let $\theta = \{t_i\}_{i=1}^{k}$ be the prefix of τ of length $|\theta| = k > c$, which is also a steady and feasible schedule.

By Lemma 3, for θ, there exists a multiset $(\mathcal{P}, m)$ of runs of length k. Since $k > c$, by Lemma A, every run $\varrho \in \mathcal{P}$ contains a rule r which is a self-loop, that is, $r.from = r.to$.

Construct a set $\mathcal{P}'$ of runs of length $k-1$, such that every $\varrho' \in \mathcal{P}'$ is obtained from some $\varrho \in \mathcal{P}$ by removing one occurrence of a self-loop rule. Given a run $\varrho' \in \mathcal{P}'$ of length $k-1$, denote by $\mathcal{P}_{\varrho'}$ the set of runs $\varrho \in \mathcal{P}$ of length k such that ϱ' was obtained by removing exactly one occurrence of a self-loop rule in ϱ. For every i, where $0 < i < k$, and $\varrho \in \mathcal{P}_{\varrho'}$, it holds that either $\varrho[i] = \varrho'[i]$, if the self-loop rule in run ϱ is at position $j > i$, or $\varrho[i+1] = \varrho'[i]$ otherwise. Construct a multiplicity function $m' : \mathcal{P}' \to \mathbb{N}$, such that for every $\varrho' \in \mathcal{P}'$, we have $m'(\varrho') = \sum_{\varrho \in \mathcal{P}_{\varrho'}} m(\varrho)$.

The multiset $(\mathcal{P}', m')$ defines a schedule $\theta' = \{t'_i\}_{i=1}^{k-1}$ of length $k-1$. We now show that θ' is feasible, and that it has the same origin and goal as θ.

To show that θ' is feasible, we show that for every i, for $1 < i < k$, and every $\ell \in \mathcal{L}$, we have $\sum_{r.to=\ell} t'_{i-1}(r) = \sum_{r.from=\ell} t'_i(r)$. By the definition of $\mathcal{P}_{\varrho'}$,

$$\begin{aligned}\sum_{\varrho'[i].from=\ell} m'(\varrho') &= \sum_{\varrho[i].from=\ell} m(\varrho) + \sum_{\varrho[i+1].from=\ell} m(\varrho) \\ &= \sum_{\varrho[i-1].to=\ell} m(\varrho) + \sum_{\varrho[i].to=\ell} m(\varrho) = \sum_{\varrho'[i-1].to=\ell} m'(\varrho')\end{aligned}$$

which implies $\sum_{r.from=\ell} t'_i(r) = \sum_{r.to=\ell} t'_{i-1}(r)$.

To show that θ and θ' have the same origin, we will show that for every $\ell \in \mathcal{L}$, we have $o(t_1).\kappa[\ell] = o(t'_1).\kappa[\ell]$. Let $\ell \in \mathcal{L}$ be a location. Without loss of generality, suppose that there is one run $\varrho^* \in \mathcal{P}$ with $\varrho^*[1].from = \ell$ and $\varrho^*[1].from = \varrho^*[1].to$, such that the self-loop rule $\varrho^*[1]$ was removed in order to obtain ϱ'. Thus, we have that $\varrho'[1].from = \varrho^*[2].from$. As all the other runs remain unchanged in the first rule, we have that

$$\sum_{\varrho'[1].from=\ell} m'(\varrho') = \sum_{\substack{\varrho[1].from=\ell \\ \varrho \neq \varrho^*}} m(\varrho) + m(\varrho^*) = \sum_{\varrho[1].from=\ell} m(\varrho)$$

which implies $\sum_{r.from=\ell} t'_1(r) = \sum_{r.from=\ell} t_1(r)$, and hence $o(t'_1).\kappa[\ell] = o(t_1).\kappa[\ell]$.

To show that θ and θ' have the same goal, we follow similar reasoning.

Observe that the goal of θ is the origin of the transition t_{k+1} in the schedule τ. Since θ and θ' have the same goal configurations, we can append $t'_k = t_{k+1}$ to the schedule θ' and obtain a new schedule τ'. The following holds for the schedule τ':

- it is feasible, since θ' is feasible and $t'_k = t_{k+1}$ is applicable in the goal of θ';
- it is steady, because $o(t_1) = o(t'_1)$ and $o(t_{k+1}) = o(t'_k)$, hence the contexts of t'_1 and t'_k are equal, since τ is steady. The steadiness of τ' follows from this and Proposition B;
- $|\tau'| = |\theta'| + 1 = k = |\tau| - 1$
- τ and τ' have the same origin, as $o(t_1) = o(t'_1)$;
- τ and τ' have the same goal, as $g(t_{k+1}) = g(t'_k)$. □

C Specifications

We call *correct* a process that is non-faulty, and *obedient* a process that is either correct, or performs a send omission, that is, a process that works correctly on the receiving side.

For the benchmarks described in Section 7, we check the following properties:

- *Unforgeability.* If no obedient process sets `v` to 1 initially, then no obedient process sets `accept` to true.
- *Validity.* A decided value is an initial value of some process.
- *Agreement*, for consensus. No two processes decide on different values.
- *k-Agreement*, for k-set agreement. At most k different values are decided.

Given a path of bounded length, we check for violations of the above mentioned properties, by constructing a reachability query as follows:

- for *Unforgeability*, we check that the initial configuration satisfies $\#\{\text{V1}\} = 0$, and the last configuration satisfies $\#\{\text{AC}\} > 0$.
- for *Validity*, we check for every value $i \in V$, whether the initial configuration satisfies $\#\{\text{VI}\} = 0$ and the last configuration satisfies $\#\{\text{VI}\} > 0$.
- for *Agreement* and *1-Agreement*, we check that the last configuration satisfies $\#\{\text{V0}\} > 0 \wedge \#\{\text{V1}\} > 0$.
- for *2-Agreement*, we check that the last configuration satisfies $\#\{\text{V0}\} > 0 \wedge \#\{\text{V1}\} > 0 \wedge \#\{\text{V2}\} > 0$.